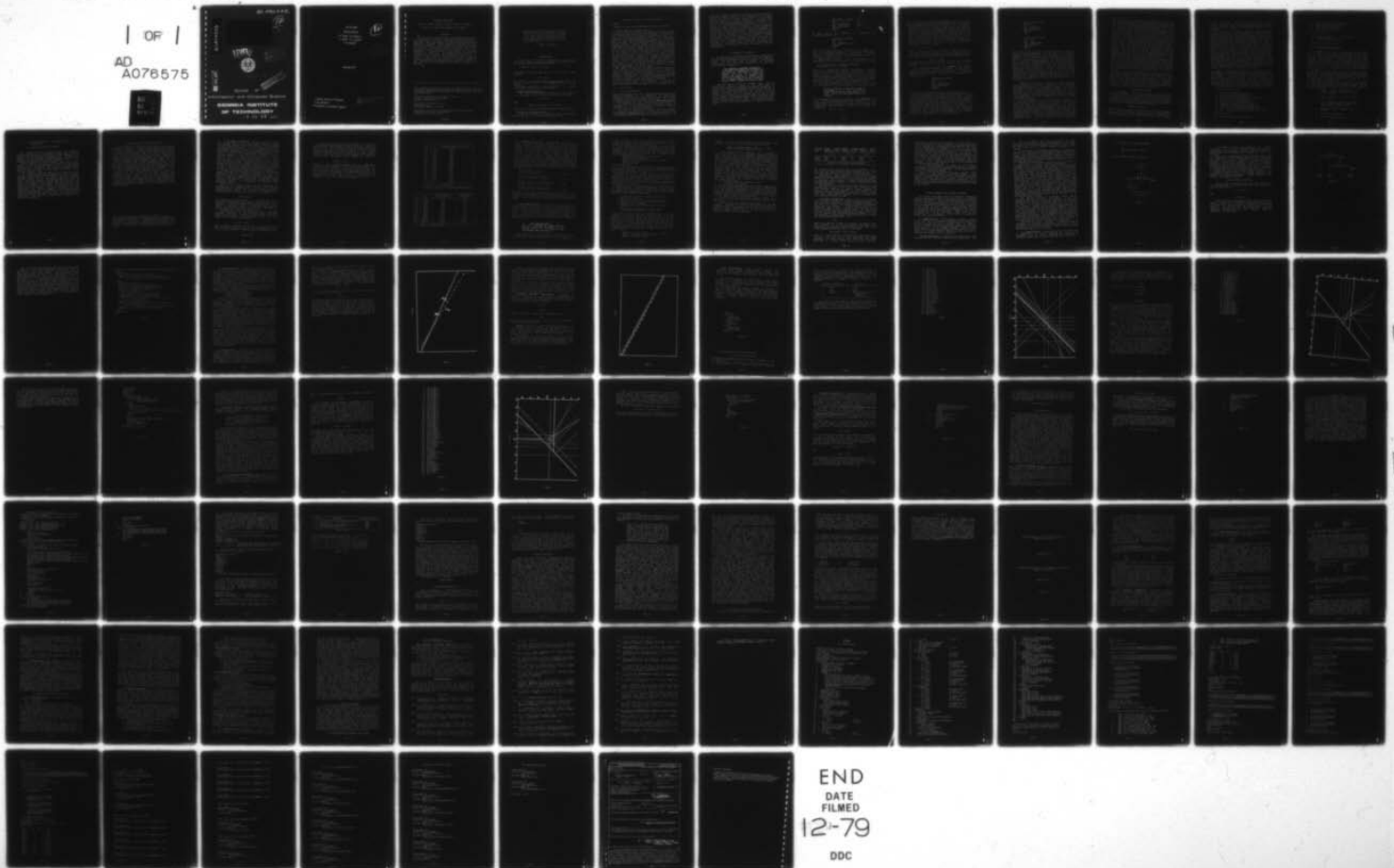
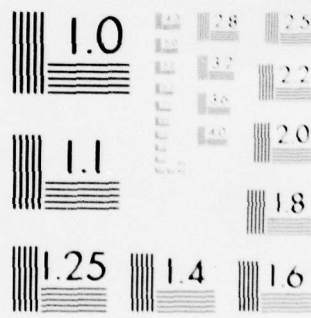


AD-A076 575 GEORGIA INST OF TECH ATLANTA SCHOOL OF INFORMATION A--ETC F/6 9/2
MUTATION ANALYSIS.(U)

UNCLASSIFIED SEP 79 A T ACREE , T A BUDD , R A DEMILLO N00014-79-C-0231
GIT-ICS-79/08 ARO-15950.4-A-EL NL

| OF |
AD
A076575





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

ARO 15950.4-A-EL

AD A 076575

12

LEVEL 14

DDC
RECEIVED
NOV 9 1979
E



See 1473 in back

DDC FILE COPY

This document has been approved
for public release and sale; its
distribution is unlimited.

School of

Information and Computer Science

GEORGIA INSTITUTE

OF TECHNOLOGY

9 11 07 067

GIT-ICS-79/08

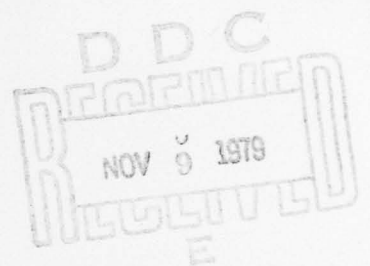
MUTATION ANALYSIS

A.T. ACREE* R.A. DEMILLO*

T.J. BUDD** R.J. LIPTON***

F.G. SAYWARD**

12



SEPTEMBER 1979

* GEORGIA INSTITUTE OF TECHNOLOGY

** YALE UNIVERSITY

*** UNIVERSITY OF CALIFORNIA, BERKELEY

This document has been approved
for public release and sale; its
distribution is unlimited.

✱

1

2

1

3

2



A

✱

1

2

3

PAGE - 1 -

Mutations are seldom spectacular. Those mutants that are startlingly different from their parents tend not to survive long, either because the mutation renders them unable to function normally, or because they are rejected by those who sired them.

Robert Silverberg

1. INTRODUCTION

A major goal of software engineering is to discover an efficiently testable property of programs, say PROP, so that for all programs P the following holds:

if PROP(P), then P is correct. (1)

By **correct** one usually means that for all possible input values x,

$$P^*(x) = f(x),$$

where $P^*(x)$ is the function computed by program P and f is a function which specifies the **intended** behavior of the program.

Prominent examples of such properties are program verification and testing for correctness:

Program Verification [Man]

Let A and B be predicates so that A(x) is true when x is in the domain of the function f and B(y) is true when

$$y = f(x).$$

Then

PROP(P) if and only if $\vdash A\{P\}B$

can be used to define the predicate PROP in proposition (1).

Testing for correctness [LMW]

Let D be a subset of all possible input to the program P, and say that D is a reliable test data set if

$P(x)=f(x)$ for all x in D implies $P*=f$.

Clearly,

$\text{PROP}(P)$ if and only if $P^*(D)=f(D)$ for some reliable D

can also be used as a property for (1).

A major stumbling block in such systematizations as these has been that the conclusion of proposition (1) is so strong that, except for trivial classes of programs P , $\text{PROP}(P)$ is bound to be formally undecidable [How1]. Given this state of affairs, program verification has turned to techniques which do not require universal applicability. It has not been clear what the corresponding course should be for program testing, however. There is an undeniable tendency among practitioners to relegate testing to completely ad-hoc techniques: one creates tests that seem to capture the essence of the program, observes the execution of the program on those tests and makes a conclusion about the correctness of the program based on the results of the observations. This strategy seems to be too undisciplined [DLS1]. More systematic techniques attempt to augment a programmer's intuition by yielding quantitative information about the degree to which a program has been tested (see [Good] for a current survey) -- such coverage measures attempt to give the tester an inductive measure of confidence that $\text{PROP}(P)$ has been determined. We will discuss several of these methods rather more fully in the sequel.

The reader should note that these techniques generally rely in one way or another on proposition (1) -- they attempt by inductive or deductive means to allow a tester to conclude correctness. But correctness is a very strong property, comprehending for instance mathematical equality of infinite functions. It is rather unlikely that efficient means can be found to make such powerful inferences.

There is another path to take, however. It is not so well travelled because it is less scenic. We propose to weaken considerably the conclusion of (1), to replace it by:

- i. P is correct
- or
- ii. P is "pathological",

where "pathological" will have a well-defined meaning, which roughly corresponds to P possessing an empirically determined characteristic which places it outside the range of programs which can be treated in this way. The testing technique determined in this way, we call **mutation analysis**.

In carrying out this plan we will of course have to sacrifice some of the elegance of the techniques based on instances of (1), but we hope that this defect is balanced by the efficacy of mutation analysis.

The sequel is organized as follows. We first present the basis of mutation analysis, relying as much as possible on **observable assumptions** about the programming process. We then describe the systems which have been constructed for

conducting mutation analysis of Fortran and Cobol programs. We present examples typical of our experience with these systems by means of several "experiments". Included among these experiments will be some evidence for believing that mutation analysis is useful in detecting a wide variety of errors (via the coupling effect introduced in [DLS1]). In Section 6, a case study is presented of the use of mutation analysis to detect errors in a production system program; it is shown in this study how test data can be strengthened to locate and remove subtle errors. Section 7 discusses the relationship of program mutation to error seeding and logic circuit fault detection. A step in the mutation analysis process involves the detection of certain kinds of program equivalence; Section 8 contains a complete discussion of this equivalence problem, suggesting some efficient algorithms for automatically detecting the appropriate equivalences. The paper closes with three nonobvious applications of the technique to issues of concern in software engineering.

2. MUTANTS OF A PROGRAM

In [DLS1], we introduced data produced by Youngs [You] that strongly hinted that the errors that are most likely to be made in the programming process are simple, classifiable errors. We have been lead to attempt the following generalization, which is used so frequently in our work that we have given it a name:

The Competent Programmer Assumption

A COMPETENT PROGRAMMER, AFTER COMPLETING THE ITERATIVE PROGRAMMING PROCESS AND DEEMING THAT HIS JOB OF DESIGNING, CODING AND TESTING IS COMPLETE, HAS WRITTEN A PROGRAM THAT IS EITHER CORRECT OR IS ALMOST CORRECT IN THAT IT DIFFERS FROM A CORRECT PROGRAM IN "SIMPLE" WAYS.

Precisely what is meant by "simple" will occupy a considerable amount of space in this paper, but the intuitive content of the competent programmer assumption is simply that competent programmers do not write programs at random; if the program produced is not correct, it is a program with bugs and can be edited into correct form by finding and fixing the bugs. Suppose that the task at hand is to design a Fortran program to compute the (Euclidean) magnitude of an N-dimensional vector X in a Cartesian coordinate system with fixed origin. Then the subroutine P1 certainly could have been produced by a competent programmer.

```

SUBROUTINE P1(X,MAG)
MAG = 1
DO 1 I = 1,N
MAG = MAG+X(I)**2
1 MAG = SQRT(MAG)
RETURN
END.

```

We would question the competence of a programmer who produced subroutine P2:

```

SUBROUTINE P2(X,MAG)
MAG = X(1)
DO 1 I = 1,N
1 MAG = MAX(X(I), MAG)
RETURN
END.

```

There is no reasonable sense in which P2 is a "buggy" version of the program asked for. P1 can easily be debugged, but P2 is not even a program of the same kind -- it is so radically incorrect that its incorrectness should be discovered by other means.

Suppose that we now try to inject this assumption into proposition (1) and try to discover a property PROP so that:

if P is written by a competent programmer (2)
and PROP(P) then P is correct.

This is a considerable change. Proposition (1) in its original form treats a program as a random object. Proposition (2) on the other hand attempts to exploit something special about the programming process (e.g., that a data processing manager expects in response to the specifications for a personnel system, something like a personnel system; perhaps incorrect, inefficient or sloppy, but more like a personnel system than, say, a missile guidance system).

To be more specific: we are after a testing method that addresses the following version of correctness testing.

Given a program P written by a competent programmer, find a test data set for which P works correctly by which we can infer that P is, with high probability, correct.

Test data which meets this criterion, we call **adequate** test data. Under the competent programmer assumption it is easy to derive some simple properties that adequate test data should have. We can observe a community of programmers and in principle classify the errors they tend to make into categories

E₁, E₂, ..., E_k.

We are free to observe the programmers for as long as we wish and make whatever specialized assumptions we wish about the programming task they will be called upon to perform. Therefore it is in principle possible to gain whatever degree of confidence we desire that among the k classifications we have countenanced the errors most likely to be made by this particular community. Given a program P to test in this setting, we must derive an adequate set of test data, D , for P . If P is incorrect, we will never be able to find an adequate set; indeed, the point of testing P is to find a set of test data that calls attention to the fact that P is incorrect. If P is correct, however, adequate D should at least convince us that P does not contain the errors most likely to be made.

Let

P_1, P_2, \dots, P_m

differ from P only in each containing a single error chosen from one of the error categories. Then an adequate set of test data D should at least provide the following assurance. For each P_j which is not equivalent to P ,

$$P^*(D) \neq P_j^*(D)$$

In other words for each of the most likely errors, it should be possible to show that P does not contain that specific error.

Each of the P_i 's is said to be a **mutant** of the program P . The competent programmer assumption states that a program is assumed to be either correct or a mutant of a correct program. For example, in the problem of computing magnitudes of N -vectors, subroutine P_1 is a mutant of the correct P below.

```
SUBROUTINE P(X,MAG)
MAG = 0.0
DO 1 I = 1,N
1 MAG = MAG+X(I)**2
MAG = SQRT(MAG)
RETURN
END
```

Subroutine P_2 , on the other hand, is not a mutant of P .

Mutation analysis is a method of eliminating the alternatives -- developing a set of test data on which P works correctly but on which all mutants of P fail (or in our suggestive terminology, "die"). Without the competent programmer assumption, there would be infinitely many mutants to consider, but even with the assumption, practice may dictate so many error types that this method is intractable. In fact, one's first reaction upon hearing of the notion is to dismiss it as an obviously intractable and therefore ridiculous idea. But by concentrating only on "simple" mutants of P the technique becomes manageable. For example, P_1 is not a simple mutant of P , but M_1 and M_2 are:

```

SUBROUTINE M1(X,MAG)
MAG = 1
DO 1 I=1,N
1 MAG = MAG+X(I)**2
MAG = SQRT(MAG)
RETURN
END

```

```

SUBROUTINE M2(X,MAG)
MAG = 0.0
SO 1 I=1,N
MAG = MAG+X(I)**2
1 MAG = SQRT(MAG)
RETURN
END.

```

The mutants we will consider arise from the single application of a mutant operator, a simple syntactic or semantic program transformation such as changing a particular instance of a relational operator to one of the remaining operators or changing the target of an unconditional transfer to another labelled target. We will also refer to mutant operators as **error operators**. The obvious objection here is that such a restriction allows one to do little more than test for typographical errors in programs, perhaps useful, but hardly worth such a fuss. As we will discuss extensively below (Section 4.3) there is an observable "coupling" of simple and complex errors so that test data that causes all nonequivalent simple mutants to die is so sensitive that "likely" complex mutants also die. The coupling of simple and complex errors implies that if P is correct for an adequate test D while M1 and M2 die, then P1 must also die on D.

Observe that mutation analysis is a valid principle (i.e., implements correctness testing) if the competent programmer assumption is valid and if the coupling of simple and complex errors is a provable effect. In practice (theoretical studies notwithstanding [BL1,BL2]) it is not necessary to show formally that these assumptions hold in order for mutation analysis to be a useful tool for testing real programs. It is sufficient to know within acceptable confidence limits when the assumptions hold and to work within those limits.

We have found that in performing mutation analysis on an incorrect program, the tester is forced to develop test data on which his program fails [BDLS]. So we are interested in building interactive systems to aid programmers and testers in performing mutation analysis -- and in so doing, evaluating the effectiveness of this approach. We pick a programming language L (Fortran, Cobol, and Lisp have been our initial choices) and -- based on prior research and other experience -- we define an appropriate set of mutant operators for L. Then we build a interactive mutation system that serves as a test harness and aids in performing

mutation analysis. Using three such systems we have for the past two years been involved in the testing of programs using mutation analysis and in experiments to discover when and why the competent programmer assumption holds and how simple errors can be coupled to complex errors.

Although the various systems we have constructed differ in certain respects, there are essential similarities. The basic design was discussed in an earlier paper [BDLS]. Briefly, the systems allow an interactive user to enter a program to be tested. The program is parsed to a convenient internal form and appropriate data files are created. The user then enters test data, executing the program on the test data in typical harness fashion to check for errors. At the point of mutation analysis, the user "turns on" a subset of the error operators and the system then creates a list of mutant description records, descriptions of how the internal form is to be modified to create the required mutant. The changes are induced sequentially and the modified internal form is interpreted, the results being compared to the original results to determine whether or not the mutant survives the execution on that data. At the completion of the pass, summary reports are presented to the user, and he is allowed several options in examining the remaining live mutants to attempt to strengthen his test data. The user may also declare mutants to be equivalent and therefore remove them from future consideration. In one of our systems this function has been partially automated with considerable improvement in performance. The issue of equivalent mutants will be discussed more fully in a later section.

Part of our early experience with mutation systems was the testing, using the first Fortran system FMS.1, of the statement scanner of FMS.1 itself. In elapsed time, the nearly 9,000 mutants were completely analyzed in six man-hours, using approximately 14 cpu minutes of a slow PDP-10 KA-10 processor running the TOPS10 timesharing operating system. A more complete description of this analysis is available in [BDLS]. We will return to the question of the efficiency of mutation analysis in the Section 4.

3. THE MUTATION SYSTEMS

3.1 Fortran. In the fall of 1977, a pilot mutation system for a subset of Fortran became operational on a PDP-10 computer at Yale University. This is the PIMS system discussed in detail in [BDLS]; in anticipation of several versions of mutation systems for several different languages we have since adopted the following naming conventions for our systems. A system is denoted by a string

`<lang>MS.<version>.`

where `<lang>` is a unique identification for the language (e.g., F for Fortran) and `<version>` is a chronological version number. Thus, PIMS is the system FMS.1. Subsequently, FMS.1 was implemented on a DEC KL-20 at Yale, and a PRIME

400 at Georgia Tech. Although the Fortran subset required by FMS.1 is restrictive, it has been large enough to permit a wide body of experience with mutation analysis to accumulate (see the experiment in [BSLS] and Section 6, for example).

The restricted language accepted by FMS.1 eventually became a bottleneck for the experimenters. Therefore, during the year 1978-1979, an expanded Fortran system, FMS.2 (the system sometimes referred to as EXPER) was constructed. FMS.2 accepts any ANSI Fortran program which does not use complex arithmetic or input/output statements (for programs which do not meet this restriction, recoding must replace input/output statements by array assignments). FMS.2 is fully operational on the DEC KL-20 at Yale and is being implemented on the VAX-11 at Berkeley. While the overall goals of the Fortran systems are similar, FMS.2 differs from FMS.1 in several important respects. FMS.1 was designed with user-oriented features in mind; it was anticipated that testers unfamiliar and unsympathetic with the system would be the primary user community. FMS.2, on the other hand, was designed primarily as an experimental device for the mutation research groups, to facilitate experiments into how mutation analysis can be integrated into the design coding and testing of multi-module programs, experiments into the sufficiency of various sets of mutant operators and for various experiments surrounding the coupling effect and the overall effectiveness of the mutation approach.

FMS.2 sessions are organized around the concept of an **experiment**. An experiment consists of a program, test data, and a subset of the error operators which may be applied to the program. The experimenter is more easily able to generate small variations in each of these elements and monitor the progress of subjects using FMS.2 to perform the mutation analysis. As with FMS.1, this system responds with summaries and reports on the number and type of mutants which remain alive, so that the user can augment his tests.

The basic set of error operators supplied by FMS.2 are

Data reference Mutations

1. Constant Replacement (by +1, -1)
2. Scalar for Constant Replacement
3. Source Constant Replacement
4. Array Reference for Constant Replacement
5. Scalar Variable Replacement
6. Constant for Scalar Replacement
7. Array Reference for Scalar Replacement
8. Comparable Array Name Replacement
9. Constant for Array Reference Replacement
10. Scalar for Array Reference Replacement
11. Array Reference for Array Reference Replacement

Operator Mutations

12. Arithmetic Operator Replacement

13. Relational Operator Replacement
14. Logical Connective Replacement
15. Unary Operator Replacement
16. Unary Operator Removal
17. Unary Operator Insertion

Statement Mutations

18. Statement Analysis (C-1 Path analysis)
19. Statement Deletion
20. Return Statement Replacement

Control Structure Mutations

21. Jump Statement Replacement
22. DO statement Replacement

3.2 **Cobol.** The design of the Cobol mutation system CMS.1 is based on the original design of FMS.1. The reader will get an idea of the way in which CMS.1 interacts with users by consulting the corresponding descriptions for FMS.1 in [BDLS]. CMS.1 accepts a simple subset of the Cobol language and supports up to ten rewindable input files and ten non-rewindable output files. This has been found to be adequate for a variety of data processing tasks and should allow the analysis of a large selection of Cobol programs. CMS.1 is currently implemented on a PRIME 400 computer at Georgia Tech.

Mutants are said to exhibit equivalent behavior if they produce the same output records as the original program. Mutants may fail by producing different output, or by a run-time error such as referencing undefined data, referencing nonnumeric data in a numeric instruction, trying to use a file unit that is not open, etc.

As might be expected, the introduction of input/output and data structuring capabilities create special problems for CMS.1 not encountered in the Fortran systems. The following are the error operators which appear to be unique to the Cobol language.

1. Move implied decimal point in numeric items one place to the left or to the right.
2. Add or subtract one from an OCCURS clause count.
3. Insert FILLER of length one between two adjacent record items; also change FILLER lengths by one.
4. Reverse adjacent elementary items in records.
5. Alter file references.
6. Switch PERFORMs and GOTOs.

7. Change ROUNDED to truncation and vice-versa.
8. Change the sense of a MOVE.

The remaining error operators include the operator replacements and control flow mutations that are described above. As primitive as this subset of Cobol appears, it is adequate for broad-based experimentation, including the analysis of many production Cobol programs supplied to the mutation research group by external sources.

CMS.1 is unique in another respect. While some module testing of FMS.1 and FMS.2 was carried out by the design teams, access to reasonable subsets of the implementation languages was limited by the concerns detailed above. CMS.1, on the other hand is being tested extensively using the FMS.2 system at Yale.

The Appendix contains essentially a script of a CMS.1 session on a production Cobol program drawn from the US Army personnel system SIDPERS. The program has been modified somewhat, mainly in the reduction of the record sizes to make a better CRT display. The program takes as input two files, representing an old backup tape and a new one. The output is a summary of the changes. The input files are assumed to be sorted on a key field. The program is 130 lines long and has 1195 mutants, of which 37 are easily seen to be equivalent to the original program. Initially ten test cases were generated to eliminate all of the nonequivalent mutants. Subsequently a subset of five test cases was found to be adequate for the task. The entire run took about 7 minutes of clock time, and 2 minutes and 45 seconds of CPU time on the PRIME 400.

4. THE COMPLEXITY OF MUTATION ANALYSIS

At first blush, it would seem that there is a severely limitative trade-off at work in the technique described in the previous section. In order to be efficient, the number of distinct mutants must be kept rather small. But the list of potential errors (rather, the list of error operators) in order to be realistic must be quite extensive. Apparently, then, if we try to constrain the number of mutants of an N statement program to some reasonable size -- say, $p(N)$, for a "small" polynomial p^* -- mutation analysis loses its effect as a realistic model of the programming process. If on the other hand we try to build into the analysis all of the possible error types which we can expect to encounter, then the number of mutants associated with an N statement program need not be bounded by any reasonable function of N .

In this section we will show how the choice of the first alternative in the tradeoff is justified. In fact, an N statement program -- on the average -- will generate only polynomially many mutants, most of which are unstable and die in the analysis stage very quickly. A "coupling effect" is invoked to save the method from only being capable of dealing with trivial errors, and we will report on some preliminary experimental evidence for our belief in the coupling effect.

*This seems reasonable. Polynomial growth in complexity in the analysis of algorithms is generally identified with computational tractability. In testing for correctness or in program verification, even subcases which are solvable tend to be of nonpolynomial complexity (usually exponential or worse).

4.1 The Number of Mutants. Youngs' data and several less widely reported but related studies [TRW,Gill] suggest very strongly that the errors that tend to occur in programs are relatively simple errors. To be precise, let us define a simple mutant as follows. Let P be a program written in a programming language defined by a grammar G, and let parse(P) be the syntax tree for P obtained by parsing P according to G. Then a 1-order simple mutant operator ER is a function mapping a parse tree T to a tree ER(T) so that T and ER(T) differ by at most one terminal node (i.e., leaf). ER(T) is said to be a simple 1-order mutant of T. Proceeding inductively, a k-order mutant is simply a k-fold iteration of 1-order mutants. In particular, notice that simple mutants do not alter the "semantic structure" of a program -- that is they do not modify the internal nodes of the parse tree. The error operators designed for the automated systems are with few exceptions simple 1-order mutants.

We will first give a heuristic analysis of the expected number of mutants of a program as a function of several size parameters. The list of mutant operators for FMS.1 and FMS.2 is relatively unsophisticated and has undergone little revision that would improve the number of generated mutants (CMS.1 by contrast has a rather more streamlined mutant generation system), so our analysis is not biased in favor of simple mutants.

First, it is possible to derive an order-of-growth expression for the number of FMS.1 mutants. Data reference replacements are accomplished by interchanging reference names occurring within the program. In a program with N statements and K distinct data references this number is

$$F(N,K) = O(K^2).$$

The reader can convince himself (cf. [Kn]) that for each of the constant and operator replacement schemes there is a constant c so that the number of generated mutants is bounded by cK. Therefore, $F(N,K)$ is the dominant term, and the number of generated mutants is in the worst case quadratic in the number of distinct data references.

Observations of typical programs lead to an even more favorable estimation of the expected number of mutants generated under FMS.2. In programs that are not maliciously dense (for an example of such a dense program see [LS]) $F(N,K)$ is more closely approximated by

$$F(N,K) = O(NK)$$

while in typical programs, such as those discovered by Knuth [Kn] the data references tend to be so sparsely distributed that the rate of growth is usually closer to quadratic in N:

$$F(N,K) = O(N^2).$$

In generating mutants of Cobol programs, it is possible to more nearly approach linear growth, since the number of data reference interchanges is limited by syntactical redundancies. In fact, an analysis similar to the one carried out above gives the worst case estimate for the expected number of mutants for a Cobol program as the number of data division lines multiplied by the number of procedure division lines. For typical Cobol programs this estimate is

$$C(N,K) \ll N^2.$$

Figures 1 and 2 show mutant growth rates for a sampling of Fortran and Cobol programs. Notice that in both cases (except for the variation in small Fortran programs) the estimates given above are generous upper bounds on the observed number of mutants. In experiments using CMS.1, we have found the average growth rate for "production" Cobol programs to be more nearly linear in the product of procedure division lines and K than quadratic in N.

N	² N	Average Number of Mutants
12	144	2508
13	169	307
14	196	427
16	256	360
17	289	390
24	576	2666
26	676	649
28	784	3213
30	900	1209
33	1089	12116*
34	1156	3361
36	1296	1085
42	1764	1057
45	2025	1658
65	4225	1514
66	4356	2425
71	5041	2817
98	9604	8424
123	15129	8838

Figure 1. Fortran Mutants

* Sample contains the outlier described in [LS].

N	² N	No. Procedure * No Data Div Lines	Total Mutants Generated
57	3249	576	370
64	4096	789	679
73	5329	756	78
74	5476	800	235
75	5625	837	225
78	6084	918	376
99	9801	1674	377
102	10404	1806	715
111	12321	2115	740
143	20449	3330	628
170	28900	5184	1195
453	205209	46803	14639
670	448900	92964	50983

Figure 2. Cobol Mutants

4.2 Mutant Instability. Even though the number of mutants generated by these methods is expected to grow rather slowly as a function of program size, a user may be somewhat wary of proffering so many executions of the same program -- it would seem that an execution of 10,000 mutant programs on a set of test data may require as much time as the running time of the longest instance multiplied by 10,000! In fact, that is not the case. A mutant seldom runs to completion; rather, mutant programs tend to be rather unstable, dying by executing "illegal" statements which are trapped and which cause premature termination of the programs. So, it is our experience that mutation analysis of even moderately large programs is possible using only modest machine requirements. The following statistics are derived from Fortran program analysis using the FMS.1 and FMS.2 systems.

Average number of test cases mutants remain live	1.75
Average total mutant executions per session (units = $F(N,K)$)	2.00
Average fraction of nonequivalent mutants killed by first test case	68%
Average execution time of live mutant (percent of original test)	75%

Although the speed with which mutants can be eliminated is a function of the capabilities of the human tester, it is our experience that somewhat more than 30% of the remaining live mutants are killed by each test case, yielding rapid convergence.

4.3 The Coupling Effect. Using only the mutant operators defined above, it would seem likely that a program that had been successfully subjected to mutation analysis might still contain some complex errors, errors which are not explicit mutants of the program and are not distinguished by the test data. In [DLS1], we proposed a "coupling effect" which asserted the existence of significant classes of programs for which such omissions are rare; briefly stated, the coupling effect asserts:

The Coupling Effect
 TEST DATA ON WHICH ALL SIMPLE MUTANTS
 FAIL IS SO SENSITIVE TO CHANGES IN THE
 PROGRAM THAT IT IS LIKELY THAT ALL COM-
 PLEX MUTANTS MUST ALSO FAIL.

Note that there is no claim that the coupling effect is a provable phenomenon in a mathematical sense; indeed, there are very simple counterexamples to it. It is however, a

useful principle that can be observed to hold for broad classes of programs. We come therefore to consider on what evidence we believe in the coupling effect.

First, we know that there is a provable coupling effect for certain restricted models of computation. In [BL1], the following was proved: Let P be a complete decision table program (i.e., one missing no actions or conditions), and let P evaluate correctly on test data that is adequate under the following mutant operators

- Replace any condition by "don't care"
- Complement any condition
- Replace any "don't care" by "yes" and "no"
- Delete any action
- Add any action;

then P is correct.

It has also been conjectured that a provable coupling effect can be exhibited for several other formally interesting classes of programs, such as pure Lisp functions and linear recursive schemes (cf [BL2]).

Second, there is a great variety of observational evidence for the coupling effect. Investigators using the DAVE test data generation system at the University of Colorado, for example, have reported that even using a restricted set of error operators the ability to detect simple errors is oftentimes useful in insuring against more complex errors [OF1,OF2].

Third, there is a growing experimental understanding of the coupling effect in functioning programs. We give here an example of the empirical evidence. The subject program is Hoare's FIND program [Hoa]. As described in [DLS1], FIND was used in the following experiment.

1. A test data set of 49 cases was derived and shown to be adequate.
2. The test data set from 1 was heuristically reduced to a set of 7 test cases which also turned out to be adequate.
3. Random simple k-order mutants were selected ($k > 1$).
4. The higher order mutants of step 3 were executed on the reduced test data set.

It would be evidence against the coupling effect if it was possible to randomly generate very many higher order non-equivalent mutants on which the reduced test data set behaved in a manner indistinguishable from FIND. Notice that Step 2 biases the experiment against the coupling effect since it removes the man-machine orientation of mutation analysis. We concentrated first on the case $k=2$, reasoning that the larger the value of k, the more one violates the competent programmer assumption, with the following results:

Number of 2-order mutants	21,100
Number indistinguishable from FIND	19
Number equivalent to FIND	19.

However, a more limited analysis of still higher order mutants, still failed to reject the coupling effect:

Number of k-order mutants ($k \geq 2$)	1,500
Number indistinguishable from FIND	0.

A major defect in this experiment can be brought to light by considering the following conceptual basis for error coupling. Just as the competent programmer assumption states that programs are not written at random, the coupling effect is implied by the fact that program statements are not composed at random; indeed, there is considerable flow and sharing of information between statements of a program, so that a change to one portion of a program is likely to have observable, albeit subtle, effects on its global context. Now for the problem with this experiment: the k-order mutants are chosen randomly and by independent drawings of 1-order mutants. Therefore the resulting higher-order mutant is very unstable and subject to quick failure. The experiment should also be conducted when the higher-order mutants contain subtly related errors. To this end, the experiment was repeated using the following replacement for step 3:

3': Randomly generate correlated k-order mutants of the program.

In Step 3', correlated means that each of the k applications of 1-order mutant operators will be related in some way to all of the preceding applications, all affecting the same line, for example. As before, if a program is successfully subjected to mutation analysis on a test data set, then the coupling effect asserts that the correlated k-order mutants are also likely to fail on the test data.

In addition to FIND, we use the program STKSIM which maintains a stack and performs the operations clear, push, pop, and top.

Figure 3 contains a summary of the results of the experiment. Although, much careful experimentation under more stringent statistical analyses must be carried out, there is probably enough information to conclude that there is a meaningful sense in which errors are coupled by an appropriate choice of error operators.

PROGRAM NAME	NUMBER GENERATED	NUMBER ALIVE	NUMBER GENERATED	NUMBER ALIVE	NUMBER GENERATED	NUMBER ALIVE
	k = 2		k = 3		k = 4	
FIND	3000	2	3000	0	3000	0
STKSIM	3000	3	3000	0	3000	0

Figure 3. Correlated k-order Mutants

The results are for the most part self explanatory. All of the live correlated k-order mutants described in the table have been shown equivalent by rather simple arguments.

Although we have attempted no thorough statistical analyses of these experiments, the size of the samples (nearly 50,000 combined correlated and uncorrelated mutants) is certainly large enough to sustain statistically significant conclusions assuming a variety of underlying models and distributions.

Less formal but nevertheless striking evidence is of the "testimonial" variety. Since 1976 we have conducted mutation analysis sessions on perhaps several hundreds of Fortran, Cobol, and Lisp programs. So many instances of the coupling of simple and complex errors have been observed over such a wide range of programs that it is likely there is an observable effect at work.

4.4 Reducing Complexity. Even with all of the foregoing reduction techniques, current technology places the bounds of practicality for monolithic programs somewhere in the 5,000 to 10,000 line range for Fortran and somewhat higher for Cobol programs. Even this must be treated as an optimistic upper limit -- certainly the technique is not easy to apply at the 5,000 statement level. A speculative but not unjustifiable technique is to use Monte Carlo techniques to sample from large populations of mutants. A simple argument to support such an analysis can be had via the following Gedanken experiment. Let

$$f(x)$$

appear in a specific context of a program undergoing mutation analysis; if a set of test data is too weak for the program but the program is nevertheless correct, then there is an adequate set of test data, D , on which

$$[f(x)]*(D) \neq [f(x')]*(D),$$

where x' is some specified data reference replacement mutation of x . But x and x' in these expression are BOUND variables; it only matters that they refer to distinct positions of a state vector which has been specially

constructed to exhibit the inequality. In other words it is important that we are able to "explain" with test data why x is an argument of f , but perhaps less important that we be able to explain why the argument is not x' or any other specific alternative. But this can be accomplished by sampling from enough alternative choices x' to insure that identities that we are observing are not mathematical. If the functions involved are at all well-behaved algebraically then algebraic identities can be discerned in this way (see [DL] for simple cases). In one experiment, mutation analysis on only 10 percent of the total mutant population resulted in test data strong enough to kill 95 percent of the entire mutant population.

If reliable patterns can be found by such sampling techniques then the range of programs which can be analyzed is expanded by an order of magnitude. We anticipate reporting on this research elsewhere.

There is an obvious method which will further reduce the amount of time needed to process mutants. Since mutants, once generated are entirely independent entities, copies of mutant description records may be distributed among several computers for parallel execution. It is feasible to decrease running times by amounts dependent only on the amount of computer resources one is willing to invest in the analysis.

5. ERROR OPERATORS FOR CLASSES OF ERRORS

Of course the whole point of program testing and therefore mutation analysis is to detect errors in programs that are not correct. So far we have given no evidence that mutation analysis is a useful tool in this regard. In this, and in the following section, we will indicate our current state of knowledge in this regard. First, we will describe a wide class of error types and show by example how the error operators which are currently implemented are useful in detecting errors of those types. Second -- in the following section -- we will describe a case study of the uncovering of a resistant, complex error in a production system using mutation analysis.

5.1 Simple Errors. If the program contains a simple error, then one of the mutants generated by the system will be correct. The error will be discovered when an attempt is made to eliminate the correct program since its behavior will be correct but the program being tested will give differing results. If the program contains simple k -order errors that are relatively independent and each error is exposed by a single mutant, then the errors will also be detected (see Section 6 for an example).

5.2 Dead Statements. As described by Huang [Hua], many programming errors manifest themselves in "dead code", that

is, source statements that are unexecutable or, more seriously, give incorrect results regardless of the data presented. Such errors may persist for weeks or even years if the errors lie in rarely executed portions of the program.

It is therefore a reasonable first goal in testing a program to insist that each statement be executed at least once. Typical methods for achieving this goal include for example the insertion of instruction counters into straight line segments of the program, so that a non-zero vector of counters indicates that the instrumented statements have all been executed at least once.

During mutation analysis, the goal outlined above will be viewed from a slightly different perspective. If a statement cannot be executed, then clearly we can change the statement in any way we want, and the effects of the changes will not be noticeable as the program runs -- in particular the altered program will not be distinguishable in its output behavior from the original one. There is, however, a mutant operator which draws the tester's attention to this situation in a more economical way. Among the mutants are those which replace in turn the first statement of every basic block by a call to a routine which aborts the run when it is executed. Such mutations are extremely unstable since any data which causes the execution of the replaced statement will also cause the mutant to produce incorrect results and hence to be eliminated. The converse is also true. That is, if any of these mutants survives the analysis then the altered statement has never been executed. Therefore, accounting for the survival of these mutants gives important information about which sections of the program have been executed.

This analysis shows why apparently useful testing heuristics can lead one astray. For example, it has been suggested [Ham] that not executing a statement is equivalent to deleting it, but this discussion shows how such a strategy can fail. A statement can be executed and still serve no useful purpose. Suppose that we replace every statement by a convenient NO-OP such as the Fortran CONTINUE. The survival or elimination of such mutants gives more information than merely whether or not the statement has been executed. It indicates whether or not the statement has any observable effect upon the output. If a statement can be replaced by a NO-OP with no observable effect, then it can indicate at best that machine time is wasted in its execution (possibly a design error) and at worst a much more serious error.

Insuring that every statement is executable is no guarantee of correctness [GG,How1]. Predicate errors or coincidental correctness may pass undetected even if every statement is successfully executed. We will return to these error types later in this section.

5.3 Dead Branches. It has been noted (see [Hua]) that an improvement over simply analyzing the execution of statements can be had by analyzing the execution of branches, attempting to execute every branch at least once.

For example, the program segment

```
A;  
IF(<expression>) THEN B;  
C;
```

has the flowchart shown in Figure 4.

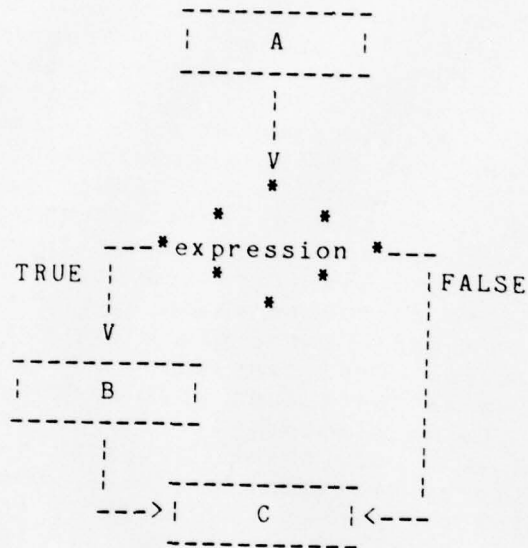


Figure 4.

All statements A,B and C can be executed by a single test case. It is not true however that in this case all branches have been executed. In this example the empty else clause branch can be bypassed even though A,B and C are executed.

However, the requirement that every branch be traversed can be restated: every predicate must evaluate to both TRUE and FALSE. The latter formulation is used in mutation analysis. There are error operators to replace each logical expression by boolean constants. Like the statement analysis mutations described above, these mutations tend to be unstable and are easily eliminated by almost any data. If these mutants survive, they point directly to a weakness in the test data which might shield a possible error.

Mutating each relation or each logical expression independently actually achieves a stronger test than that achieved by the usual techniques of branch analysis. For consider the compound predicate

IF(A.LE.B.AND.C.LE.D)THEN ...

Simple branch coverage requires only two test cases to test the predicate. But suppose that the test points for the covering test are

$A < B$ and $C < D$

and

$A < B$ and $C > D$.

These points have the effect of only testing the second clause. This kind of analysis fails to take into account the hidden paths [DLS1] implicit in compound predicates (see Figure 5). In testing all the hidden paths, mutation analysis requires at least three points to test the predicate, corresponding to the branches $(A > B, C > D)$, $(A < B, C > D)$, and $(A < B, C < D)$.

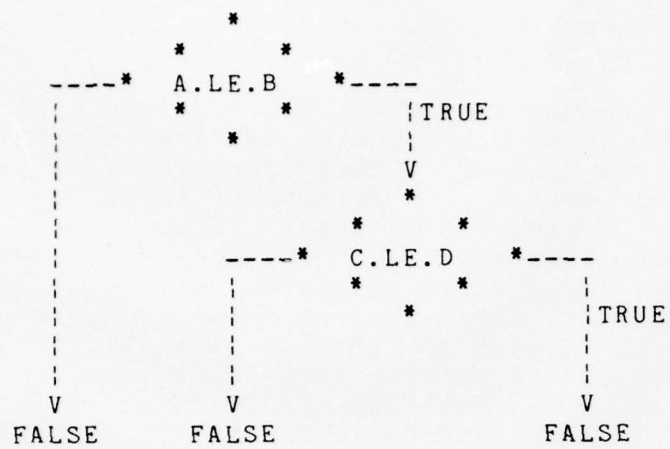


Figure 5.

As a more concrete example, consider the program shown in Figure 6. This program is adapted from [Gell] and was studied in [OW]; it is intended to calculate the number of days between two given dates. The predicate which determines whether a year is a leap year is incorrect. Notice that if year the year is divisible by 400 (i.e., if $\text{year REM } 400 = 0$) it is necessarily divisible by 100 (ie, $\text{year REM } 100 = 0$). Therefore the logical expression formed by the conjunction of these clauses is equivalent to the second clause alone. Alternatively the expression $\text{year REM } 100 = 0$ can be replaced by the logical constant TRUE and the resulting mutant is equivalent to the original program. Since it is not obvious what the programmer had in mind, the error is discovered. Notice also that mutation analysis shows that the assignment $\text{daysin}(12) := 31$ is redundant and can be removed from the program.

```

PROCEDURE calendar(INTEGER VALUE day1,month1,day2,month2,year);
BEGIN
  INTEGER days
  IF month2=month1 THEN days=days2-days1
    COMMENT if the dates are in the same month, then
      we can compute the number of days directly;
  ELSE
    BEGIN
      INTEGER ARRAY daysin(1..12)
      daysin(1):=31;daysin(3):=31;daysin(4):=30;
      daysin(5):=31;daysin(6):=30;daysin(7):=31;
      daysin(8):=31;daysin(9):=30;daysin(10):=31;
      daysin(11):=30;daysin(12):=31;
      IF ((year REM 400)=0) OR
        ((year REM 100)=0 and (year REM 400)=0)
        THEN daysin(2):=28 ELSE daysin(2):=29;
      COMMENT set daysin(2) according to whether or not
        year is leap year;
      days:=day2+(daysin(month1)-day1);
      COMMENT this yields the number of days in complete
        intervening months;
      FOR i:=month1 +1 UNTIL month2-1 DO days:=daysin(i)+days;
      COMMENT add in the days in complete months;
    END
    WRITE(days)
  END;

```

Figure 6.

5.4 Data Flow Errors. A program may access a variable in one of three ways. A variable is said to be defined if the the result of a statement is to assign a value to the variable. A variable is said to be referenced if its value is required by the execution of a statement. Finally, a variable is said to be undefined if the semantics of the language does not explicitly give any other value to the variable. Examples of the latter are the values of local storage after procedure return or Fortran DO loop indices after normal loop termination.

Following Fosdick and Osterweil [OF2] we define three types of data flow anomalies which are often indicative of program errors. These anomalies are consecutive accesses to a variable of the following forms:

1. undefined then referenced,
2. defined then undefined,
3. defined then redefined.

Anomaly 1 is almost always indicative of an error, even if it occurs only on a single path between the point at which the variable becomes undefined and its point of reference. Anomalies 2 and 3 tend to indicate errors when they are unavoidable, that is, when they occur along a cut set of the flow graph.

The second and third types of anomalies are attacked directly by mutation operators. If a variable is defined and is not used then in most cases the defining statement can be eliminated without effect (by insertion of a CONTINUE statement for instance). This may not be the case if in the course of defining the variable a function with side effects is invoked. In this case, the definition can very likely be altered in many ways with no effect on the side effect, resulting in the variable being given different values. An attempt to to remove these mutations will usually result in the anomaly being discovered.

It is more difficult to see which operators address anomalies of the first type; the underlying errors are attacked by the discipline imposed by mutation analysis. Recall that a mutation system is a large interpretive system for automatically generating and testing mutants. Whenever the value of a variable becomes undefined it is set by the interpreter to the unique constant UNDEFINED. Before every variable reference a check is performed by the interpreter to see if the variable has undefined values. If the variable is UNDEFINED the error is reported to the user, who can then take action.

5.5 Domain Errors. The notion of a domain error is due to Howden [How1]. A domain error occurs when an input value causes an incorrect path to be executed due to an error in a control statement. Domain errors are to be contrasted with computation errors which occur when an input value causes the correct path to be followed but an incorrect function of the input value is computed along that path due to an error in a computation statement. These notions are not precise

and it is difficult with many errors to decide in which category they belong.

A method of reliably uncovering domain errors is the domain strategy proposed by White, Cohen, and Chandrasekaran [WCC]. For a program containing N input variables (e.g., parameters, arrays, and I/O variables), any predicate in the program can be treated as an algebraic relationship and can thus be described by a surface in the N dimensional input space. If, as often happens, the predicate is linear, then the surface is a hyperplane. Consider a two dimensional example with input variables I and J

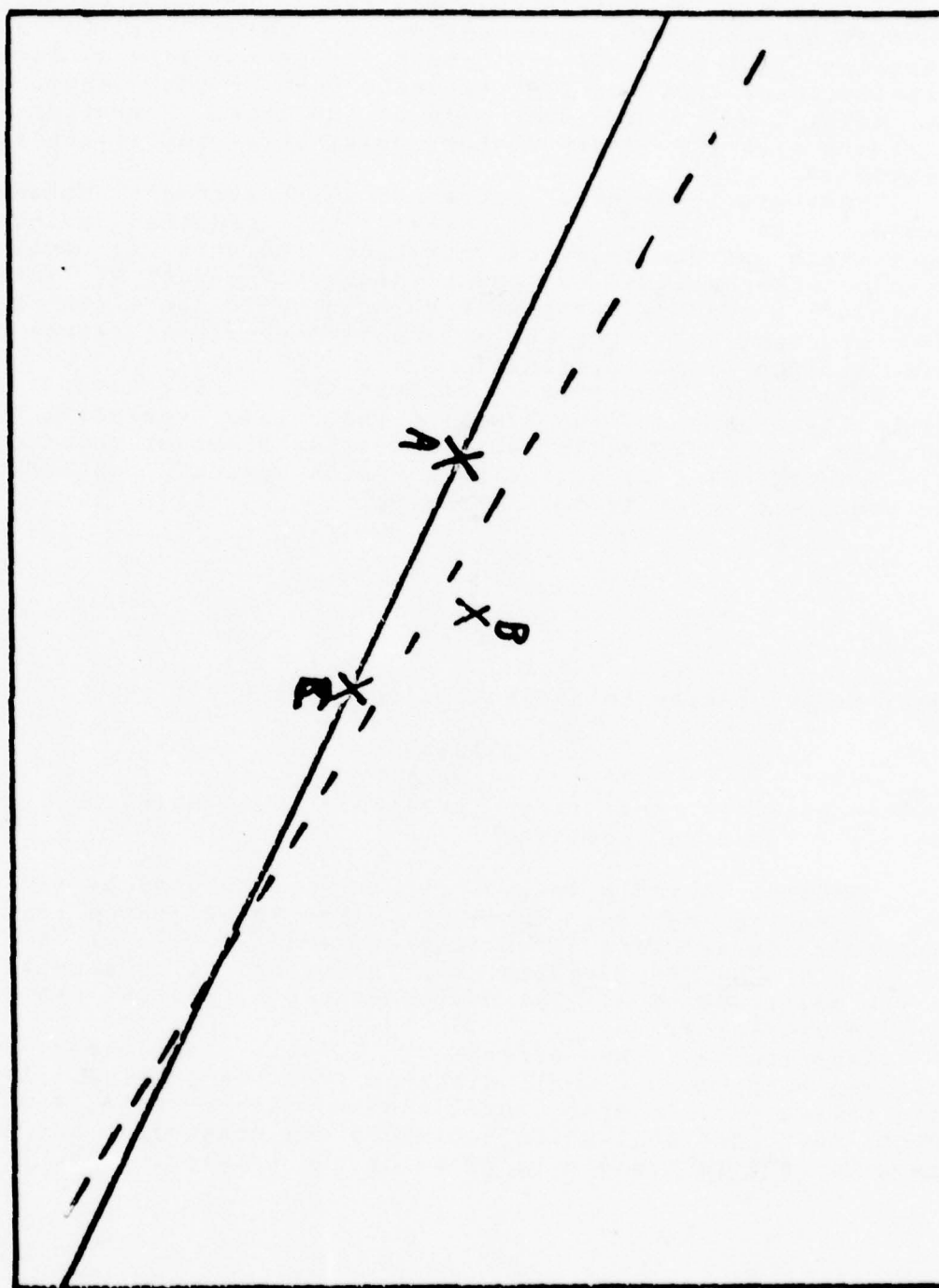
$$I+2J \leq -3.$$

The domain strategy tests this predicate using three test points, two on the line

$$I+2J=3,$$

and one point which lies off the line, but within an envelope of width $2d$ centered on the line (see Figure 7). Call these points A, B and C . If A, B , and C yield correct output, we know that the defining curve of the predicate must cut the sections of the triangle ABC . Choosing d small enough makes the chance of the predicate actually being one of these alternatives small. Therefore, even if one doesn't have complete confidence that the predicate is correct, we have gained some inductive confidence that the predicate is correct.

Figure 7



Mutation analysis also deals with the issue of domain errors. Indeed the domain strategy can be implemented using mutation once a simple observation is made: it is not necessary that points A and B both lie on the line -- it is only necessary that the line separate them or that they do not both lie on the same side of the line. Hereafter we will work with the domain strategy using this simplifying assumption.

There are three error operators which generate mutants causing the tester to generate the required points. Intuitively, we can think of mutation analysis as posing certain alternatives to the predicate in question. These alternatives require the tester to supply "reasons" (in the form of test data) why the alternative predicate cannot be used in place of the original.

Relational Operator Replacement. Changing an inequality operator to a strict inequality, weakening the operator, or changing its sense generates a mutant which can only be eliminated by a test point which exactly satisfies the predicate. For example changing

$$I+2J \leq 3$$

to

$$I+2J < 3$$

requires the tester to generate a point on the line

$$I+2J=3$$

which satisfies the first predicate but which does not satisfy the second predicate.

Twiddle. Twiddle is a unary operator denoted by ++ or --, depending on its sense. In the FMS.2 system ++a is defined to be a+1 if a is an integer and a+.01, if a is real. In the CMS.1 system, ++a is defined to be sensitive to the magnitude of a. The complementary operator --a is defined similarly.

Graphically, the effect of twiddle is to move the proposed constraint a small distance from the original line (see Figure 8). In order to eliminate these mutants, a data point must be found which satisfies one constraint but not the other and is hence very close to the original line.

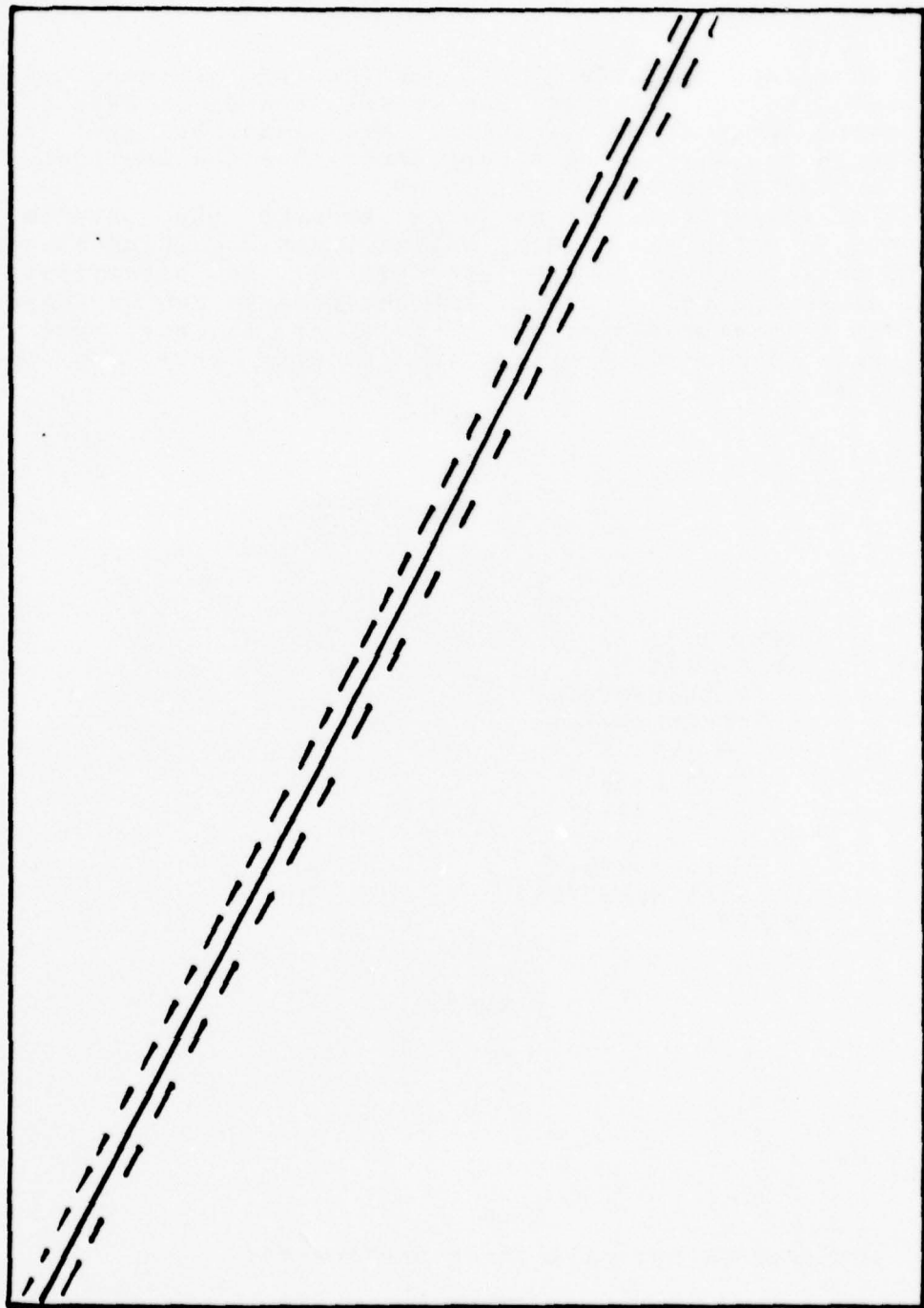


Figure 8

Other Replacements. These operators replace data references with other syntactically meaningful data references and similarly for operators. These effects are related to the phenomenon of "spoilers" which are described in 5.8.

The practical effect of considering so many alternatives is to increase the total number of data points necessary for their elimination. This leads by the domain strategy to an increased confidence that the predicate has been correctly chosen.

For comparison, let us work through the program in Figure 9, which was used by White, Cohen and Chandrasekaran [WCC] to illustrate domain strategies. No specifications are given for this program, but the program can be compared against a presumably correct version; in any case the program is useful since it involves only two input variables.

```
READ I,J;
IF I < J+1
  THEN K=I+J-1
  ELSE K=2*I+1;
IF K > I+1
  THEN L=I+1
  ELSE L=J-1;
IF I=5
  THEN M=2*L+K;
  ELSE M=L+2*K-1
WRITE M;
```

Figure 9.

The program has only three predicates:

$I < J+1$, $K > I+1$, and $I=5$.

The effect of changing the first of these is typical, so we will deal with it.

Figure 10 is a listing of all the alternatives tried for the predicate $I < J+1$. Some of these are redundant

(e.g., $++I \leq J+1$ and $I \leq --J+1$), but this is merely an artifact of the generation device; the redundancies can be easily removed (see Section 8). The alternative predicates introduced in this way are illustrated in Figure 11. The original predicate line is the heavy line. White et. al. hypothesize that the program of Figure 9 contains the errors:

statement/expression	should be
$K \geq I+1$	$K \geq I+2$
$I=5$	$I=5-J$
$L=J-1$	$L=I-2$
$K=I+J-1$	THEN IF($2 * J < -5 * I - 40$)
	THEN $K=3$;
	ELSE $K=I+J-1$;

We leave it to the reader to verify that attempting to eliminate the alternative $K \geq I+2$ necessarily ends with the discovery of the first error. Note that this is not trivial since errors 1 and 4 can interact in a subtle way. In the sequel we show how the remaining errors are dealt with.

1. IF($I \leq J$)
2. IF($I \leq J+2$)
3. IF($I \leq J+1$)
4. IF($I \leq J+J$)
5. IF($1 \leq J+1$)
6. IF($2 \leq J+1$)
7. IF($5 \leq J+1$)
8. IF($I \leq 1+1$)
9. IF($I \leq 2+1$)
10. IF($I \leq 5+1$)
11. IF($I \leq J+5$)
12. IF($-I \leq J+1$)
13. IF($++I \leq J+1$)
14. IF($--I \leq J+1$)
15. IF($I \leq -J+1$)
16. IF($I \leq ++J+1$)
17. IF($I \leq --J+1$)
18. IF($I \leq -(J+1)$)
19. IF($I \leq J-1$)
20. IF($I \leq \text{MOD}(J, 1)$)
21. IF($I \leq J$)
22. IF($I \leq 1$)
23. IF($I < J+1$)
24. IF($I = J+1$)
25. IF($\text{.NOT. } I = J+1$)
26. IF($I > J+1$)
27. IF($I \geq J+1$)

Figure 10.

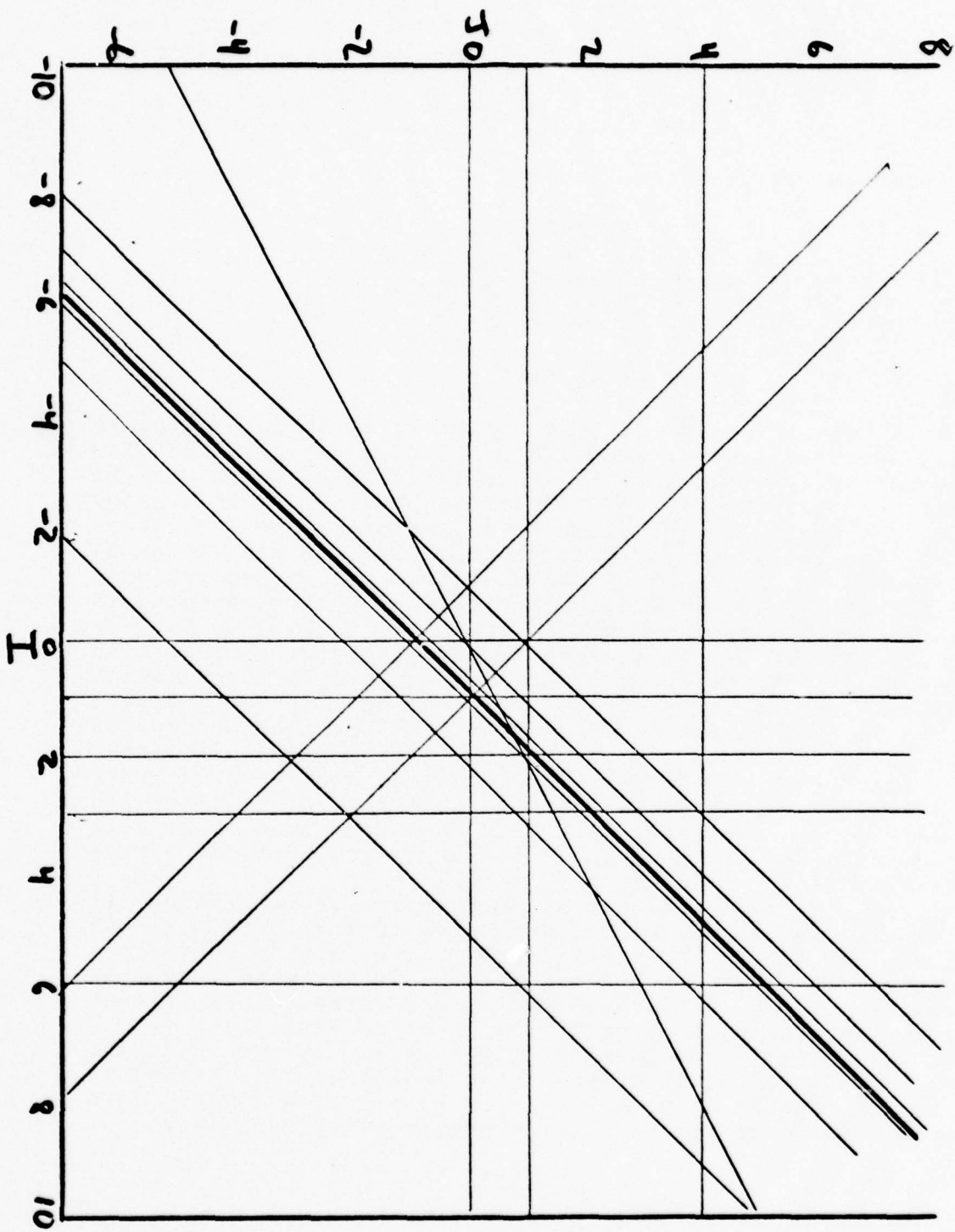


Figure 11

The introduction of the unary ++ and -- operators can be generalized in several useful ways. In addition to the twiddle operators, we consider the unary operator - and the extra-syntactic operators ABS (absolute value), -ABS (negative absolute value), and ZPUSH (zero push). Consider the statement

$$A=B+C.$$

In order to eliminate the mutants

$$A=ABS(B)+C,$$

$$A=B+ABS(C),$$

and

$$A=ABS(B+C),$$

we must generate a set of test points in which B is negative (so that $B+C$ differs from $ABS(B+C)$, C is negative, and $B+C$ is negative). Notice that if it is impossible for B to be negative then this is an equivalent mutation. That is, the altered program is equivalent to the original one. In this case, the proliferation of these alternatives can either be a nuisance or an important documentation aid, depending upon the testers' point of view. The topic of equivalent mutants will be taken up again later.

In similar fashion, negative absolute value insertion forces the test data to be positive. We use the term **domain pushing** for this process. By analogy to the domain strategy, these mutations push the tester into producing test cases where the domains satisfy the given requirements.

Zero Push is an operator defined so that $ZPUSH(x)$ is x if x is nonzero, and otherwise is undefined so that the mutant dies immediately. Hence the elimination of this mutant requires a test point in which the expression x has the value zero.

Applying this process at every point where an absolute value sign can be inserted gives a **scattering** effect. The tester is forced to include test cases acting in various positions in several problem domains. Very often, in the presence of an error, this scattering effect causes a test case to be generated in which the error is explicit.

Returning to the example in Figure 9, we can generate the additional alternatives shown in Figure 12. Figure 13 shows the domains into which these mutants push. Even this simple example generates a large number of requirements!

1. IF(ABS(I)>J+1)
2. IF(I>ABS(J)+1)
3. IF(I>ABS(J+1))
4. K=(ABS(I)+J)-1
5. K=(I+ABS(J))-1
6. K=ABS(I+J)-1
7. K=ABS((I+J)-1)
8. K=2*ABS(I)+1
9. K=ABS(2*I)+1
10. K=ABS(2*I+1)
11. IF(ABS(K)<I+1)
12. IF(K<ABS(I)+1)
13. IF(K<ABS(I+1))
14. L=ABS(I)+1
15. L=ABS(I+1)
16. L=ABS(J)-1
17. L=ABS(J-1)
18. IF(.NOT.ABS(I)=5)
19. M=2*ABS(L)+K
20. M=2*L+ABS(K)
21. M=ABS(2*L+K)
22. M=ABS(L)+2*K-1
23. M=L+2*ABS(K)-1
24. M=ABS(L+2*K)-1
25. M=ABS(L+2*K-1)

Figure 12.

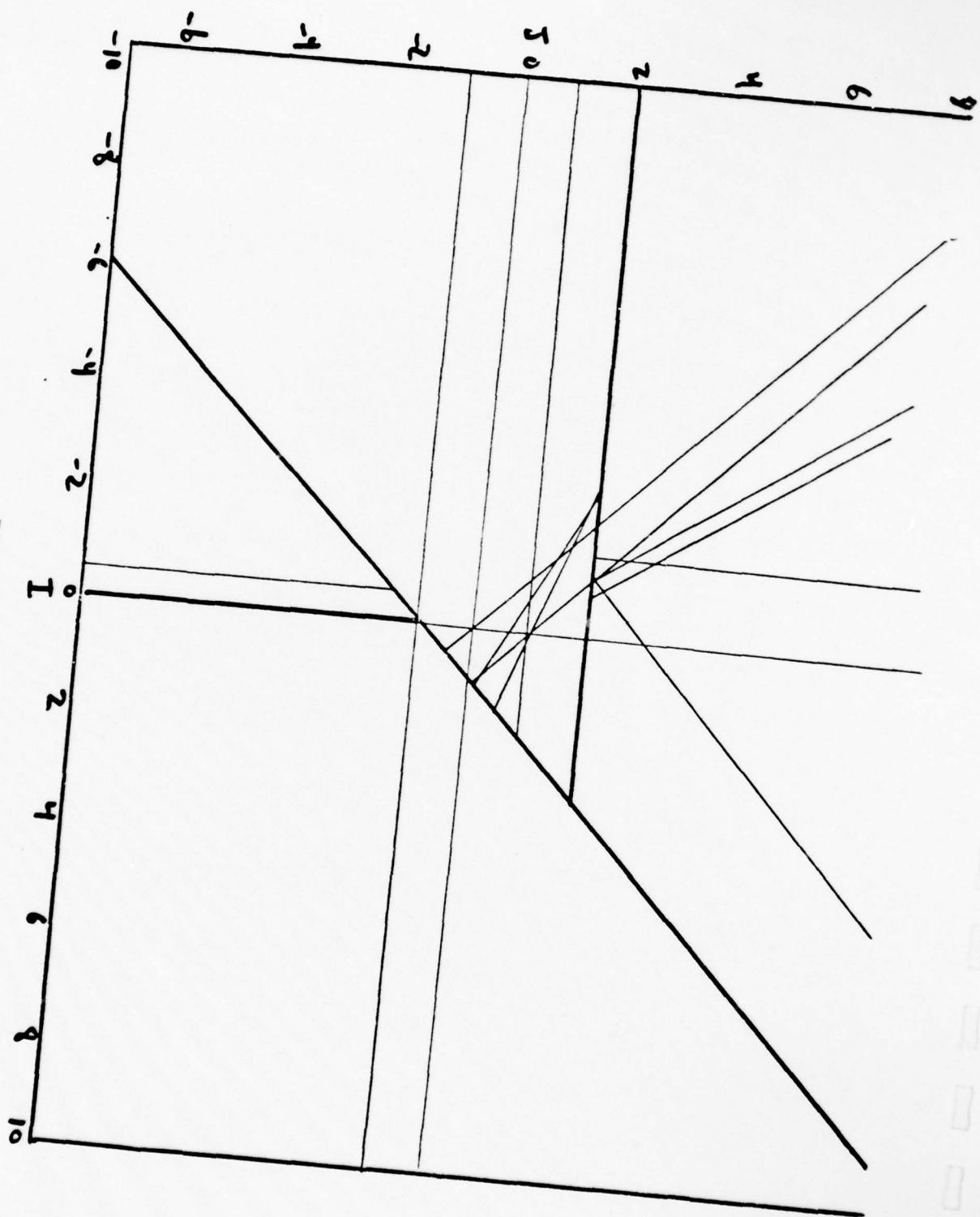


Figure 13

One effect of the error $L=J-1$ is that **any** test point in the area bounded by $I=J+1$ and $I=1$ will return an incorrect result. But this is precisely the area that mutants 8,9, and 10 push us into. So, the error could not have gone undiscovered in mutation analysis.

This process of pushing the tester into producing data satisfying some criterion is also often accomplished by other mutations. Consider the program in Figure 14, which is based on a text reformatter program by Nauer [Nau] and which has been previously studied in the program testing literature [GG].

```

alarm:=FALSE
bufpos:=0;
fill:=0;
REPEAT
  incharacter(cw);
  IF cw=BL or cw=NL THEN
    IF fill+bufpos ≤ maxpos THEN
      outcharacter(BL);
    ELSE
      BEGIN
        outcharacter(NL);
        fill:=0;
        FOR k:=1 STEP 1 UNTIL bufpos DO outcharacter(buffer[k]);
        fill:=fill+bufpos;
        bufpos:=0
      END
    ELSE
      IF bufpos = maxpos THEN alarm:=TRUE;
      ELSE BEGIN
        bufpos:=bufpos+1;
        buffer[bufpos]:=cw
      END
    END
  UNTIL alarm or cw=ET

```

Figure 14.

Consider the mutant which replaces the first statement `fill:=0` with the statement `fill:=1`. The effect of this mutation is to force a test case to be defined in which the first word is less than `maxpos` characters long. This test case then detects one of the five errors originally reported in the program [GG]. The surprising thing is that the effect of this mutation seems to be totally unrelated to the statement in which the mutation takes place!

5.6 Special Values. Another form of test which has been introduced by Howden [How2] is **special values** testing. Testing of special values is defined in terms of a number of "rules". For example:

1. Every subexpression should be tested on at least one test case which forces the expression to be zero.
2. Every variable and every subexpression should take on a distinct set of values in the test case.

The relationship between the first rule and domain pushing (via zero values mutations) has already been discussed. The second rule is undeniably important. If two variables are always given the same value then they are not acting as free variables and a reference to the first can be uniformly replaced with a reference to the second. But this is also an error operator and the existence of these mutations enforces the goals of Rule 2.

A slightly more general method of enforcing Rule 2 might use the following device. A special array exactly as large as the number of subexpressions to be computed in the program is kept. Each entry in this array has two additional tag bits which are initialized to their low values indicating that the array is uninitialized. As each subexpression is encountered in turn, the value at that point is recorded in the array and the first tag bit is set. Subsequently, when the subexpression is again encountered if the second tag is still off the current value of the expression is compared against the recorded value. If these values differ the second tag is set to high values; otherwise no change is made. By counting those expressions in which the second tag bit is low and the first is high one can infer which expressions have not had their values altered over the test case. Mutations could be constructed to reveal this. This technique is similar to one used in a compiler system by Hamlet [Ham]

5.7 Coincidental Correctness. The result of evaluating a given test point is **coincidentally correct** if the result matches the intended value in spite of a computation error. For example, if all our test data results in the variable `I` taking on the values 2 and 0, then the computation

$$J = I * 2$$

may be coincidentally correct if the intended calculation was

$J = I^{**2}$.

The problem of coincidental correctness is really central to program testing. Every programmer who tests an incorrect program and fails to find the errors has really encountered an instance of coincidental correctness. In spite of this, there has been no direct assault on the problem and some authors have gone so far as to say that the problems of coincidental correctness are intractable [WCC].

In mutation analysis, coincidental correctness is attacked by the use of **spoilers**. Spoilers implicitly remove from consideration data points for which the results could obviously be coincidentally correct -- this "spoils" those data points. For example by explicitly creating the mutation

$J = I * 2 \Rightarrow J = I^{**2}$

we spoil those test cases for which $I=0$ or $I=2$ are coincidentally correct and require that at least one test case have an alternative value.

Continuing with the example of Figure 9, Figures 15 and 16 show the spoilers and their effects associated with the statement $M = L + 2 * K - 1$. Notice that a single spoiler may be associated with up to four different lines depending on the outcome of the first two predicates in the program. In geometric terms, the effects of the spoilers are that within each data domain for each line there must be at least one test case which does not lie on the given line. In broad terms, the effects of this are to require that a large number of data points for which the possibilities of coincidental correctness are very slight.

1. $M = (L + 1 * K) - 1$
2. $M = (L + 3 * K) - 1$
3. $M = (I + 2 * K) - 1$
4. $M = (J + 2 * K) - 1$
5. $M = (K + 2 * K) - 1$
6. $M = (L + 2 * J) - 1$
7. $M = (L + 2 * I) - 1$
8. $M = (L + 2 * L) - 1$
9. $M = (L + I * K) - 1$
10. $M = (L + J * K) - 1$
11. $M = (L + K * K) - 1$
12. $M = (L + L * K) - 1$
13. $M = (L + 2 * K) - I$
14. $M = (L + 2 * K) - J$
15. $M = (L + 2 * K) - K$
16. $M = (L + 2 * K) - L$
17. $M = (1 + 2 * K) - 1$
18. $M = (2 + 2 * K) - 1$
19. $M = (5 + 2 * K) - 1$
20. $M = (L + 2 * 1) - 1$
21. $M = (L + 2 * 2) - 1$
22. $M = (L + 2 * 5) - 1$
23. $M = (L + 5 * K) - 1$
24. $M = (-L + 2 * K) - 1$
25. $M = (L + -2 * K) - 1$
26. $M = (L + 2 * -K) - 1$
27. $M = (L + 2 * --K) - 1$
28. $M = -(L + 2 * K) - 1$
29. $M = -((L + 2 * K) - 1)$
30. $M = (L + 2 + K) - 1$
31. $M = (L + 2 - K) - 1$
32. $M = (L + \text{MOD}(2, K)) - 1$
33. $M = (L + 2 / K) - 1$
34. $M = (L + 2 ** K) - 1$
35. $M = (L + 2) - 1$
36. $M = (L + K) - 1$
37. $M = L - 2 * K - 1$
38. $M = (\text{MOD}(L, 2 * K)) - 1$
39. $M = L / 2 * K - 1$
40. $M = L * 2 * K - 1$
41. $M = L ** (2 * K) - 1$
42. $M = L - 1$
43. $M = (2 * K) - 1$
44. $M = L + 2 * K + 1$
45. $M = \text{MOD}(L + 2 * K, 1)$
46. $M = (L + 2 * K) / 1$
47. $M = (L + 2 * K) * 1$
48. $M = (L + 2 * K) ** 1$
49. $M = (L + 2 * K)$
50. $M = 1$

Figure 15.

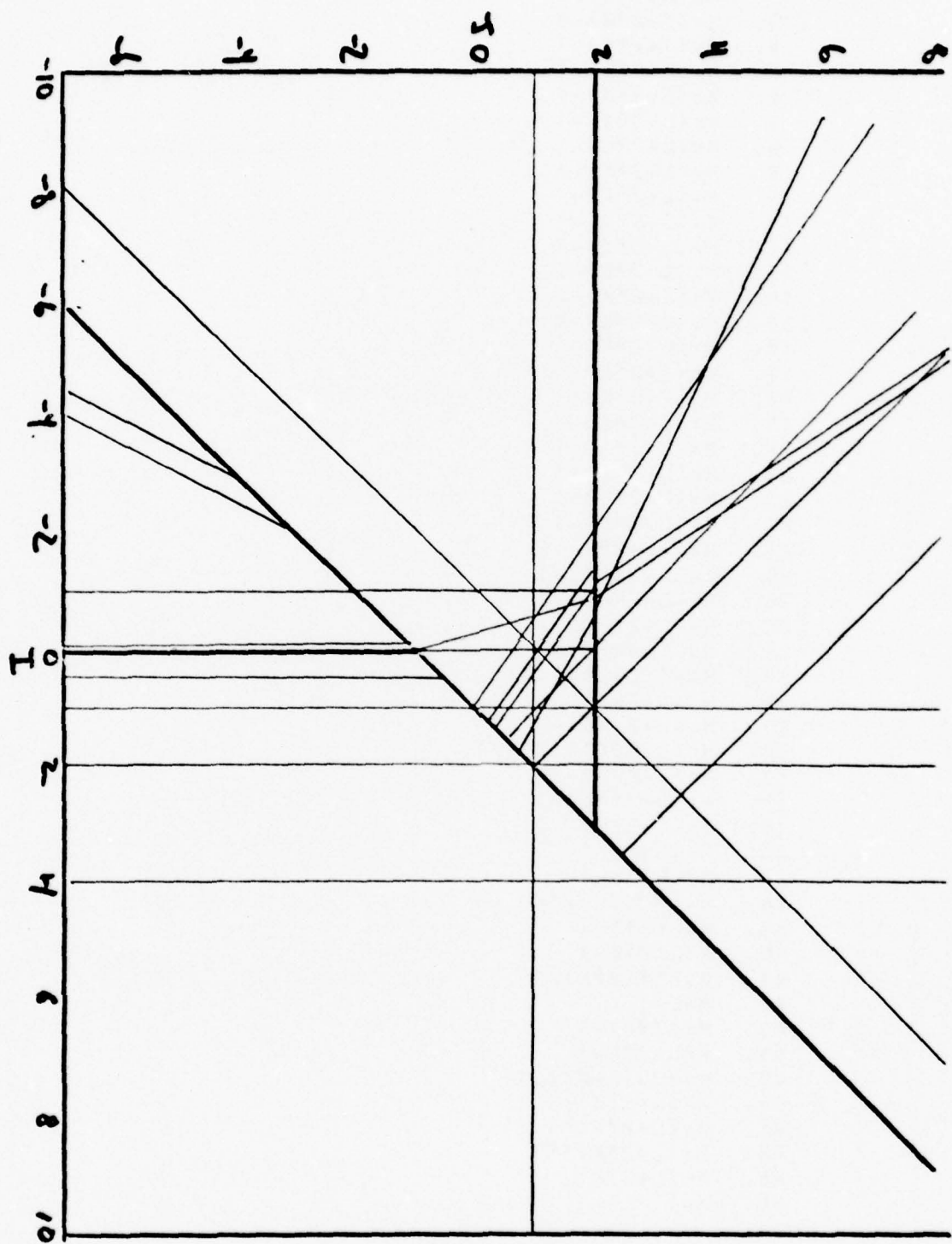


Figure 16

Often the fact that two expressions are coincidentally the same over the input data is a sign of a program error or of poor testing. The sorting program of Figure 17 is from [Wir], and it performs correctly for a large number of input values. If, however, the statements following the IF statement are never executed for some loop iteration it is possible for R3 to be incorrectly set and an incorrectly sorted array will result.

By constructing the mutant which replaces the statement

$$a(R1) := R0 \Rightarrow a(R1) := a(R3)$$

it is clear that there are two ways of defining R0, only one of which is used in the test data. This exposes the error.

```
FOR R1=0 BY 1 TO N BEGIN
  R0:=a(R1);
  FOR R2=R1+1 BY 1 TO N BEGIN
    IF a(R2)>R0 THEN BEGIN
      R0:=a(R2);
      R3:=R2
    END
  END
  R2:=R0;
  a(R1):=R0;
  a(R3):=R2
END;
```

Figure 17

5.8 Missing Path Errors. A program contains a missing path error if a predicate is required which does not appear in the subject program, causing some data to be computed by the same function when an altogether different function of the input data is called for. The definition is due to Howden [How2]. Such missing predicates can really be the result of two different problems, however, so we might consider the following alternative definitions.

A program contains a **specificational missing path error** if two cases which are treated differently in the specifications are incorrectly combined into a single function in the program. On the other hand, a program contains a **computational missing path error** if within the domain of a single specification a path is missing which is required only because of the nature of the algorithm or of the data involved.

An example of a specificational error is the fourth error from the example in Section 5.5. Although this error might result from a specification there is nothing in the code itself which could give any hint that the data in the range

$$2 * J < 5 * I - 40$$

is to be handled any differently than shown in the program.

As an example of the second class of path error consider the subroutine shown in Figure 18, which is adapted from [KP]. The input consists of a sorted table of numbers and an element which may or may not be in the table. The only specification is that upon return

$$X(\text{LOW}) \leq A \leq X(\text{HIGH})$$

and

$$\text{HIGH} \leq \text{LOW} + 1.$$

A problem arises if the program is presented with a table of only one entry, in which case the program diverges.

In the specifications there is no clue that a one-entry table is to be treated any differently from a $k > 1$ entry table. The algorithm makes it a special case.

```

SUBROUTINE BIN(X,N,A,LOW,HIGH)
INTEGER X(N),N,A,LOW,HIGH
INTEGER MID
LOW=1
HIGH=N
6  IF(HIGH-LOW-1)7,12,7
12 RETURN
7  MID=(LOW+HIGH)/2
   IF(A-X(MID))9,10,10
9   HIGH=MID
   GO TO 6
10  LOW=MID
   GO TO 6
END

```

Figure 18.

Computational missing path problems are usually caused by requirements to treat certain values (e.g., negative numbers) differently from others. When this occurs, data pushing and spoiling often lead to the detection of the errors. In the example under consideration here an attempt to kill either of the mutants

IF(HIGH-LOW-1)12,12,7

or

MID=(LOW+HIGH)-2

will cause us to generate a test case with a single element.

Since mutation analysis -- like all testing techniques -- deals mainly with the program under test, the problem of dealing with specificational missing path errors appears to be considerably more difficult. Under the Competent Programmer Assumption and the Coupling Effect, however, a tester who has access to an "oracle" for the program specifications can assume that the mutants cover all program behavior! So by consulting the specifications the tester can detect missing paths by noting incomplete behavior and thus uncover any missing paths. But since the assumptions of a competent programmer and coupling are statistical and since it may be infeasible to check for incomplete behavior, the chances of detecting such missing paths are not certain.

To see this failure, consider the missing path error from section 5.5. It is possible to generate test data which is adequate but which fails to detect the missing path error because there is no oracle to consult for completeness of behavior. This appears to be a fundamental limitation of the testing process. Unlike, say, program verification, program testing does not require uniform a priori specifications; rather we only ask that the tester be able to judge correctness on a case-by-case basis. It is our view that the only way to attack these problems is to start with a core of test cases generated from specifications, independent of the subject program. This core of test cases can then be augmented to achieve stronger goals. We note that some preliminary work on generating test data from specifications has already been reported [GG,OW].

5.9 Missing Statement Errors. By analogy with missing path errors, a **missing statement error** is defined by a statement which should appear in the program but which does not. It is not clear that the techniques of statement analysis can be used to uncover these errors. In fact, it is rather surprising that mutation analysis -- a technique which is directly oriented toward examining the effect of a modification to a statement -- can be used to detect missing statements at all!

To see how this can be accomplished, consider the program shown in Figure 19. This program accepts a vector V of length N and returns in MPSUM the value

$$V(i)+V(i+1)+\dots+V(N)$$

where $j=i-1$ is the smallest index such that $V(j)$ is strictly positive. In degenerate cases, $MPSUM=0$ is returned.

There is a missing RETURN statement which should follow the IF statement. The effect of the error is to cause undefined behavior when the vector V is uniformly nonpositive (undefined, since DO loop variables are of indeterminate value after normal completion of the loop).

A simple mutation of MPADD is the transformation

```
DO 1 I=1,N ==> DO 1 I=1,N+1.
```

This mutant fails only when the loop executes $N+1$ times. In this case all elements of V are nonpositive and the original program fails, so eliminating this mutant uncovers the error. But even after adding the return statement, MPADD will still be incorrect due to a missing path error. We leave it to the reader to discover the error by considering the mutant

```
DO 1 I=1,N ==> DO 1 I=1,N-1.
```

```

SUBROUTINE MPADD(V,N,MPSUM)
INTEGER V(N),N,MPSUM
MPSUM = 0
DO 1 I=1,N
1  IF(V(I).GT.0)GO TO 2
2  M=I+1
DO 3 I=M,N
3  MPSUM=MPSUM+V(I)
RETURN
END

```

Figure 19.

6. A CASE STUDY

To see the effect of mutation analysis on a tester who is attempting to locate and remove program errors, it is worthwhile to examine a debugging session for a program that is not known beforehand to be "testable". This case study differs from previous mutation dialogs which we have reported [DLS1,DLS2,LS] in that our previous reports dealt with programs strongly believed to be correct, for which mutation analysis was used as a tool to increase our confidence in the program's correctness. The subject program to be discussed here is known to contain at least one "resistant" error; the error had resisted all of the usual debugging techniques such as selective traces and statement instrumentation. Hence, mutation analysis is used here not as a test data **evaluator** but as a tool for systematic debugging and, perhaps just as importantly, as a convenient run time environment for Fortran subroutines.

The subject program is a routine called NXTLIV. It is a key routine in the CMS.1 system and can be considered a production program for purposes of testing. NXTLIV accepts as input the identifying number of a mutant of a given type and returns the number of the next live mutant, as indicated by bit maps of the live mutants. The bit maps are in general too large to fit in an internal array so they must be paged from a random access disk file as needed. Similar maps of the dead mutants and equivalent mutants are also stored. The subject program is shown in Figure 20.

```

      SUBROUTINE NXTLIV(MTYPE,MUTNO)
C FIND THE NEXT LIVE MUTANT AFTER THE MUTNOth OF TYPE MTYPE
C RETURN THIS VALUE IN MUTNO.
C A VALUE OF ZERO RETURNED MEANS NO MUTANTS OF THAT TYPE
  REMAIN ALIVE.
      NOLIST
$INSERT ICS057>CPMS.COMPAR>SYSTEM.PAR
$INSERT ICS057>CPMS.COMPAR>MACHINE.SIZES.PAR
$INSERT ICS057>CPMS.COMPAR>FILENM.COM
$INSERT ICS057>CPMS.COMPAR>TSTDAT.COM
$INSERT ICS057>CPMS.COMPAR>MSBUF.COM
      LIST
      INTEGER MTYPE,MUTNO
      INTEGER I,J,K,L,WORD,BIT
      LOGICAL ERR
C      CALL TIMER1(33)
C ASSUME THAT THE RECORD CONTAINING THE LIVE BIT MAPS FOR
C MUTNO IS ALREADY PRESENT, UNLESS MUTNO=0.
      K=BPW-1
C CHECK TO SEE IF WE ARE AT THE END OF A PHYSICAL RECORD
      IF(MUTNO.EQ.0)TO TO 1
      IF(MOD(MUTNO,K*MSFRS).EQ.0)GO TO 24
      GO TO 10
1      CALL REARAN(MSFILE,LIVBUF,MSFRS,LIVPTR,ERR)
      IF(ERR)CALL ABORT('(NXTLIV) ERROR IN MUTANT STATUS FILE',36)
      CALL REARAN(MSFILE,EQUBUF,MSFRS,EQUPTR,ERR)
      IF(ERR)CALL ABORT('(NXTLIV) ERROR IN MUTANT STATUS FILE',36)
      CALL REARAN(MSFILE,DEDBUF,MSFRS,DEDPTR,ERR)
      IF(ERR)CALL ABORT('(NXTLIV) ERROR IN MUTANT STATUS FILE',36)
      CHANGD=.FALSE.
      WORD=1
      BIT=2
      GO TO 20
10     WORD=MOD((MUTNO)/(K),MSFRS)+1.
      BIT=MOD(MUTNO,K)+2
20     DO 22 J=WORD,MSFRS
      L=LIVBUF(J)
      IF(L.NE.0)GO TO 23
      MUTNO=MUTNO+K
      IF(MUTNO.GT.MCT)GO TO 40
      GO TO 22
23     DO 21 I=BIT,BPW
      MUTNO=MUTNO+1

      IF(MUTNO.GT.MCT)GOTO40
      IF(AND(L,2**(BPW-I)).NE.0)GO TO 30
21     CONTINUE
      BIT=2
22     CONTINUE
24     OF(.NOT.CHANGD)GOTO 25
C SAVE OLD RECORDS
      CALL WRTRAN(MSFILE,LIVBUF,MSFRS,LIVPTR,ERR)
      CALL WRTRAN(MSFILE,EQUBUF,MSFRS,EQUPTR,ERR)
      CALL WRTRAN(MSFILE,DEDBUF,MSFRS,DEDPTR,ERR)
C NEED TO GET NEXT RECORDS
25     LIVPTR=LIVPTR+MSFRS

```

```

      EQUIPTR=EQUIPTR+MSFRS
      DEDPTR=DEDPTR+MSFRS
      GO TO 1
30    GO TO 9999
40    MUTNO=0
      IF(.NOT.CHANGD)GO TO 9999
C SAVE OLD RECORDS
      CALL WRTRAN(MSFILE,LIVBUF,MSFRS,LIVPTR,ERR)
      CALL WRTRAN(MSFILE,EQUBUF,MSFRS,EQUIPTR,ERR)
      CALL WRTRAN(MSFILE,DEDBUF,MSFRS,DEDPTR,ERR)
9999  CONTINUE
C     CALL TIMER2
      RETURN
      END

```

Figure 20.

Since FMS.1 provides a more user-oriented environment than FMS.2, NXTLIV was tested using FMS.1. To adapt to the smaller Fortran subset of FMS.1, some modifications had to be made. Since FMS.1 does not accept PARAMETER statements the parameters BPW and MSFRS (from the \$INSERT blocks) were replaced with typical values. Allowances had to be made for the unsupported CALL and the random I/O routines. The two TIMER calls were ignored. Integer arithmetic was used to simulate the remaining features. To facilitate testing several parameters are entered as explicit formal parameters.

FMS.1 first asks for the parameter values:

```
MUTNO = 0
MCT = 6 (MCT is the total number of mutants of current type)
CHANGD = 0
LIVBUF(1)=LIVBUF(2)=7
LIVBUF(3)=LIVBUF(4)=0
NLB(1)=...=NLB(4)=0 (NLB is the next live buffer. It should be
                      transferred to LIVBUF for use immediately)
LLB(1)=...=LLB(4)=0 (LLB is the last live buffer)
```

Once the data is entered the system executes NXTLIV on the test points and responds:

PARAMETERS ON OUTPUT

```
MUTNO = 0
LIVBUF(1)=0
LIVBUF(2)=0
LIVBUF(3)=0
LIVBUF(4)=0
LLB(1)=0
LLB(2)=0
LLB(3)=0
LLB(4)=0
CHANGD=0
```

THE RAW PROGRAM TOOK 41 STEPS TO EXECUTE THIS TEST CASE

The output MUTNO=0 signifies that the end of the live mutant map for this type has been reached. The tester then informs the system that NXTLIV has worked correctly for this test case. The first type of mutant to be investigated by the tester is SAN (Statement Analysis), which replaces statements by traps. The FMS.1 mutation report for this run is as shown below.

POST RUN PHASE

```
NUMBER OF TEST CASES = 1      NUMBER OF MUTANTS = 44
NUMBER OF LIVE MUTANTS= 23    PCT. ELIMINATED MUTANTS = 47.73
```

Examination shows the mutants shown in Figure 21(a) to be still live.

In attempting to kill these mutants the tester generates the testcases 2 and 3 (see Figure 21(b)).

line	statement	has been changed to
16	IF((MUTNO/12)*12.EQ.MUTNO)GO TO 24	TRAP
17	GO TO 10	TRAP
32	WORD=((MUTNO/3)-4*((MUTNO/3)/4))+1	TRAP
34	BIT=MUTNO-3*(MUTNO/3)+2	TRAP

Figure 21(a)

test	MUTNO	MCT	CHANGD	LIVBUF	NLB	LLB
case				1 2 3 4	1 2 3 4	1 2 3 4
2	1	6	0	7 7 0 0	0 0 0 0	00 00 00 00
3	10	20	1	1 3 0 0	7 7 0 0	99 99 99 99
15	5	20	0	0 0 1 0	1 1 1 1	99 99 99 99

Figure 21(b).

Testcase 2 eliminates twelve of the remaining SAN mutants. Testcase 3, on the other hand produces the output

PARAMETERS ON OUTPUT

MUTNO=14
LIVBUF(1)=7
LIVBUF(2)=7
LIVBUF(3)=7
LIVBUF(4)=0
LLB(1)=1
LLB(2)=3
LLB(4)=0
LLB(5)=0

THE RAW PROGRAM TOOK 56 STEPS TO EXECUTE THIS TEST CASE.

An error has been detected; the correct output for MUTNO is 13 instead of 14. This error resulted from choosing a starting point in the middle of a word of zero bits. NXTLIV ordinarily searches the bits of each word looking for the next "1", but for efficiency a whole word is compared to zero before the search is begun. If all bits are set low, MUTNO is incremented by the word length and the next word is accessed. A correct algorithm would increment MUTNO only by the number of bits left to be examined in the word. The only way this can make a difference in the original program is for NXTLIV to be called in such away as to stop at a "1" bit in the middle of the word, which is otherwise all 0's, and then by a mutant failure or equivalence (outside the routine) to have that bit turned off before NXTLIV is called again for the next mutant to be considered. Obviously this situation is so rare that it is bound to defy haphazard debugging attempts but is none the less common enough to cause irritation in a production-sized Cobol run.

The needed fix is to replace

MUTNO=MUTNO+K

by

MUTNO=MUTNO+(K-(BIT-2)).

After eliminating all SAN mutants and turning on the remaining error operators, a total of eleven test cases killed all but 50 of 1,514 mutants, about 96.7 percent of the total. Eventually the tester's attention is directed to the mutant at line 45

BIT=2 ==> I=2.

The testcase 15 in Figure 21(b) is an attempt to eliminate this mutant. The program again fails and another error has been found. This error is also related to the test for the entire word of zeroes. By starting in the middle of a word

of zeroes, the BIT pointer is not correctly set to 2 to begin searching the next word. The correction is to replace

```
      BIT=2  
22      CONTINUE
```

by

```
22      BIT=2
```

An interesting note is that this "correction" is actually a mutation that the tester would have had to eliminate in any event, so in effect the error was uncovered by the coupling effect before it was explicitly considered.

In completing the analysis of NXTLIV the tester of course has to deal with the equivalent mutants. This subject will be discussed in more detail in a later section. The complete analysis of the corrected program required the elimination of 1,580 mutants. The corrected algorithm has since been running without known failure in CMS.1.

7. SEEDING AND FAULTS

There are two previously suggested error detection techniques which seem to bear strong resemblance to mutation analysis. They arise in different settings and the relationship of mutation analysis to both of them has been questioned in several private correspondences. One of these is the **error seeding** technique described with several applications by Gilb [Gil] and the other is **fault detection** [Cha] applied to circuit design. Mutation analysis has almost nothing in common with error seeding, but owes a great deal to fault detection work in switching theory.

The idea behind error seeding is to insert "random" errors in a program. This approach has been used in several studies of the programming and debugging process. In one experiment the seeds were used to calibrate the effectiveness of software documentation on its maintainability; in another experiment the number of errors in a program is estimated by inserting the seeds and then uncovering k errors, using the percentage of those k errors which were seeded to infer the total number of errors.

On the surface this idea seems very similar to mutation. Let us look a little more closely at the notion of "randomness" which is so crucial to the technique. First, if we inspect the results of the experiments described in [Gil], we are struck by the lack of resolution. In the first experiment described above, for example, "randomly" chosen groups of programmers were given various sets of clues about the programs to be debugged. As reported by Gilb: "Variations between individuals in homogeneously selected groups of programmers are at least 2 to 1 and up to 10 to 1." Furthermore, the interpretations consistent with the experimental results tend to be highly suspect: "The use of test data seems to be less effective than simple

source program reading."

The reason for such results is apparent in the following description given by Gilb of the statistical basis for using seeding to estimate the total number of errors in a program.

How many fish are there in a pond or a lake? Let's say that a reasonably large sample of 1000 fish are marked and then allowed to mix for a while with the total population in the pond. If we then take a new sample of 1000 fish and find that 50 of these have our markings on them, this gives us 20,000 fish as a reasonable estimate if we accept the original sample as random and the remixing of the fish as homogeneous.

This seems to be the source of the difficulty. We have strong evidence that, first, the fish tend to school in ways that are not predictable. So in order to get a truly random sample we have to know where to fish beforehand, and second, the marked fish show truly idiosyncratic tastes in picking their associations in the pond. In particular, there seems to be no way at all of insuring that the sample we obtain neither underestimates nor overestimates the original population by unpredictable amounts. In less prosaic terms the preponderance of evidence obtained through mutation analysis (see [DLS2,LS] for indicative studies) is that errors do not occur with statistical properties that make them useful for error seeding studies. Even though they may be considered the result of a stochastic process whose properties can be determined for **small well-defined aggregates**, they are in individual programs sporadic, highly non-independent, and not uniformly distributed through the code. It is precisely because the inserted errors are random that they do not relate in a regular way to the natural errors. As we have seen, it takes much care in the choice of error operators to insure that specific categories of errors are reliably detectable by mutation analysis.

A hallmark of mutation analysis is that it rests on the Competent Programmer Assumption; we explicitly assume that a program is not a random object. A program once it is created contains errors and these are fixed, deterministically located objects. In order for a statistical technique to be applicable to a given program a considerable number of a priori assumptions must be rather fully justified. It is, however, possible to design experiments on fixed populations of programs, whose properties are quantifiable, which will reveal statistical properties of such hypotheses as the Coupling Effect. But this is an entirely different issue.

To clearly draw the distinction it may be helpful not to think of the mutants as being errors, but simply as small perturbations of the program's structure. As we have seen, these perturbations have the effect of insuring that the

test data exercises the program in a thorough fashion. If the test data is sensitive to the perturbations, then one's confidence that what was written was what was intended is correspondingly increased. If on the other hand, the test data allows one to alter the program significantly without changing its apparent behavior, then one has little confidence in the test.

Finally, mutation analysis has a psycho-social aspect that error seeding cannot have. Even if error seeding worked perfectly, the assumptions which make it work would also insure that it give no information about where the remaining natural errors occur (statistical independence insures this). Mutation analysis forces a controlled reconsideration of the source code. It leads -- as we saw in the section preceeding this one -- to a situation in which the tester must consider statement x and ask himself "why does it not matter if statement x is changed to x'?" The possible answers are that statement x is in error, that it does matter but the test data does not reveal it, that x is equivalent in context to x', or that the programmer does not understand statement x and is unable to give a reason. In each situation information about the program, about the test, and about the programmer is revealed.

Fault detection experimentation is a classical technique for detecting faults in switching circuits. The crucial idea is that one systematically "faults" circuit elements and examines the input-output function of the resulting circuit by comparing it to the original circuit [Cha]. This is the key idea of mutation analysis. There are, however, some essential differences which make mutation analysis applicable on a larger scale. First, the principle use of fault detection is to check circuit deterioration, not to validate design. Second, because circuits tend not to be functionally organized the technique is exhaustive when applied to design testing (for deterioration experiments there is frequently fault data available to guide the experimenter). In essence, the approach adopted by mutation analysis is fault detection applied to systems of high functionality in the presence of the Competence Programmer Hypothesis and the Coupling Effect. This suggests that perhaps mutation analysis in its automated form can be used for circuit validation. Perhaps, although the lack of functional description at the switching element level makes it hard to avoid the exhaustive and therefore combinatorially explosive growth of the test cases. But technology has grown in an unexpected direction in the last twenty years, and the digital design techniques of today seem to be not ill-suited to mutation analysis. In preliminary hand studies to be reported elsewhere, we have used the mutation analysis approach to test micro-coded circuit designs with surprising success.

8. THE PROBLEM OF MUTANT EQUIVALENCE

Experience indicates that in production programs, the

number of equivalent mutants can vary between 2% and 5% of the total mutant count. In more finely tuned program (see, eg, our analysis of FIND in [DLS1] and Burns' analysis of sorting routines [Bur]), however it is common for source statements to appear in a particular form solely for efficiency reasons. In these program such statements can be altered without affecting the output behavior. A typical example of this behavior is beginning a loop at 2 instead of 1 or 0, so that a mutation which changes

2 ==> 1

for example, causes an extra iteration but does not alter the outcome of the looping operation. In tuned programs, the equivalent mutants can comprise as much as 10% of the total.

It is easy to show that equivalent mutant detection is a formally undecidable problem (note that equivalent mutant detection is not obviously the same problem as the general equivalence problem for program schemata [Man]). Assume a fixed programming language which is expressive enough to allow the programming of all recursive functions, and let P1 and P2 be arbitrary procedures written in the language. Since "goto" mutations are meaningful and likely mutations, consider the following program to which goto replacement has been applied.

goto L;		go to M;
L:P1;halt;	==>	L:P1;halt;
M:P2;halt;		M:P2;halt;

Clearly, these two programs are equivalent (that is, they either halt together and deliver the same output or they diverge together) if and only if P1 and P2 are equivalent, and that is undecidable for the language described above. In fact, our choice of language is needlessly complex; essentially the same proof holds for the Fortran subset accepted by FMS.1 and the Cobol subset accepted by CMS.1.

In spite of this, most equivalent mutants are stylized and rather easy to judge equivalent. This is perhaps due to the Competent Programmer Assumption: the subject program and an allegedly equivalent mutant are not chosen randomly -- in fact, they are chosen by a very careful sieving of all possible programs and the structure of this relationship should be something that one can exploit in determining mutant equivalence.

Before we proceed it may be instructive to examine a few instances of equivalent mutants which show this structure. In the analysis of SCAN (see Section 2), a relatively large number of mutants resulting from the transformation

X ==> RETURN

appear as live mutants on even very good test data. On closer examination, however, most of these reveal that

X = GO TO 90,

where statement labelled 90 is itself a RETURN. The programmer's style is to always jump to a common RETURN statement, allowing an easy "proof" of equivalence.

For a more pregnant example, let us return to the NXTLIV routine described above. A principal source of equivalent mutants in that example was the troublesome test for a word of zeroes. Its only purpose is to save the effort of looking through the words bit by bit. If the condition is the test is replaced by any identically true expression, the program runs a bit longer but is otherwise identical(see Figure 22(a)). Similarly the mutation shown in Figure 22(b), changes the performance of the program only, but this time it improves it!

IF(L.NE.0)GOTO 23 ==> IF(12.NE.0)GO TO 23
(applied at line 34)

Figure 22(a)

IF(MUTNO.GT.MCT)GOTO 40 ==> IF(MUTNO.GE.MCT)GOTO 40
(applied at line 36)

Figure 22(b).

Figure 22.

These last two examples are not accidental. Mutations of a program are remarkably similar to simple transformations that are made in code optimization; it is not surprising that some of them should turn out to be optimizing or de-optimizing transformations. Conversely, correctness preserving optimizing transformations should be applicable to detecting equivalent mutants. If this is a useful heuristic then the task of identifying equivalent mutants can be reduced to detecting those which are equivalent for an **interesting** reason.

Almost all of the techniques used in optimizing compiled code can be applied in some way to decide whether a mutant is equivalent to the subject program. Some optimizing transformations are widely applicable while others are severely limited in scope. We will give a sampling of the useful transformations. For terminology and detailed discussions see [AU,Sch].

8.1 Constant Propagation. Constant propagation involves replacing constants to eliminate run-time evaluation. A typical optimizing transformation would replace statement 3 as shown below

1	A=1		1	A=1
2	B=2	==>	2	B=2
3	C=A+B		3	C=3

There are several elegant schemes for global transformations of this form.

Constant propagation is most useful for detecting cases in which a mutant is not equivalent to the subject program; any change which can affect the known value of a variable can be detected in this fashion. The mechanism for testing equivalence of mutants using constant propagation is to compare at all points after the mutation site the constants which are globally propagated through the program. If they differ it is likely that the programs are not equivalent. The test is certain if there is a RETURN, HALT or some other exit statement in which the set of associated constants contains an output variable and if there is a path from the entry point of the program to the exit point. This is resolvable by dead code detection (see 8.6).

8.2 Invariant Propagation. Invariant propagation generalizes constant propagation by associating with each statement a set of invariant relations between data elements (e.g., $X < 0$ or $B = 1$). Although invariant propagation has met with limited applicability in compiler design, it is a powerful technique for detecting equivalent mutants, particularly those involving relational mutant operators. These operators frequently only affect an expression if it has a certain relationship to 0. For example $|x|$ changes the value of x only if $x < 0$. In the program-mutant pair

IF(A.LT.0)GOTO1		IF(A.LT.0)GOTO1
B=A	==>	B=ABS(A)

the conditional allows us to determine the invariant ($A \geq 0$) and this allows us to determine that the program and its mutant are equivalent since the absolute value of a positive number is that number.

Invariant propagation is enhanced if the propagation and testing algorithms exploit transitivity of the relations and allow the replacement of an invariant by a weaker one.

8.3 Common Subexpressions. Perhaps the most common optimization is to recognize calculations which are repeated but which can be pre-computed. For example

$$\begin{aligned} A &= X+Y \\ B &= X+Y+Z \end{aligned}$$

calculates $X+Y$ twice, but can be replaced by a program which uses a temporary variable to hold $X+Y$.

A common iterative algorithm for eliminating common subexpressions uses global analysis to associate with each statement the propagated variables, but this time partitioned into equivalence classes under the equivalence of evaluating to the same value. Since this method generates equivalent expressions not used in the program, the widest possible range of equivalent subexpressions is recognized. This is a very useful technique for dealing with mutations to assignment statements. Changing an operator changes the equivalence class of the variable to which the assignment was made. Similarly mutations which change an operand or destination in an assignment will produce changes in the equivalence classes following the assignment. Therefore, comparing the equivalence partitions can demonstrate differences between the subject and the mutation.

Consider the mutation

$$A = B+C \text{ (partition = } A; B+C) \Rightarrow A = B-C \text{ (partition = } A; B-C)$$

Comparing the partitions shows that A has a different value in the two programs.

The same ideas are used to show equivalence. If a mutation has changed part of expression E to an expression E' but E and E' are in the same equivalence class, then the mutant is equivalent.

8.4 Loop Invariants. Another common transformation removes code from inside loops if the execution of that code does not depend on the iteration of the loop. Since many mutations change the boundaries of loops techniques for recognizing this invariance is useful for detecting equivalent mutants. In those cases where the mutation either increases or decreases the code within a loop, loop invariant recognition can be used to decide whether or not the effect of the loop is changed. In the following mutation, excess code is brought within the scope of the `DO` statement.

	DO 1 I=1,10	==>		DO 2 I=1,10
	A(I)=0			A(I)=0
1	CONTINUE		1	CONTINUE
2	B=0		2	B=0

Since the assignment B=0 is loop invariant, it does not matter how many times it is executed.

8.5 Hoisting and Sinking. Hoisting and sinking is a form of code removal from loops in which code which will be repeatedly executed is moved to a point where it will be executed only once; this is accomplished by a calculus which gives strict conditions on when a block of code can be moved up (hoisted) or down (sunk).

The applications for equivalence testing are similar to the applications for loop invariants. The major difference is that hoisting and sinking applies to cases in which code is included or excluded along an execution path by branching changes. These are the sorts of changes obtained by GOTO replacement and statement deletion mutations. In these cases, we get equivalence if the added or deleted code can be hoisted or sunk out of the block involved in the addition or deletion.

An example will illustrate.

	IF(A.EQ.0)GOTO1	==>		IF(A.EQ.0)GOTO 2
	A=A+1			A=A+1
2	B=0		2	B=0
	GO TO 3			GO TO 3
1	B=0		1	B=0
3	.		3	.
	.			.
	.			.

In this example B is set to 0 regardless of whether it is assigned its value at line 1 or at line 2. The assignment to B can be hoisted as follows:

	B=0
	IF(A.EQ.0)GO TO 3
	A=A+1
3	.
	.
	.

Since both programs are thus transformed, they are equivalent.

8.6 Dead Code. Dead Code detection is geared toward identifying sections of code which cannot be executed or whose execution has no effect. Dead code algorithms exist for detecting several varieties of dead code situations. We have already used dead code analysis as a subproblem in the propagation problems above. Dead code analysis is also useful to directly test equivalence, particularly for those mutations arising from an alteration of control flow.

A typical application is to analyze the program flow-

graphs. If, for example, a mutation disconnects the graph and neither connected component consists entirely of dead statements, then the mutant cannot be equivalent. Such disconnection is possible by the mutant which inserts RETURNS in Fortran subroutines.

Another common situation involves applying mutations to sites in a program which are themselves dead code; this is the classical compiler code optimization problem: we must detect dead code since any mutations applied to it are equivalent.

Dead code analysis can also be used to show nonequivalence by using it to demonstrate that a mutation has "killed" a block of code.

8.7 Postprocessing the Mutants. Optimizing transformations can be implemented as a postprocessor to a mutation system. User experience is that it is relatively easy to kill as many as 90% of the live mutants. To the remaining 10%, an equivalence heuristic such as the rules sketched above can be applied. A more complete description of such a postprocessor is available in [BaS].

The difficulty of judging equivalent mutants from those remaining after the postprocessing stage both helps and hinders the testing process. On one hand, forcing testers and programmers to "sign off" on equivalent mutants enforces a unique sort of accountability in the testing phase of program development (see Section 9). On the other hand, particularly clever programming leads to many equivalent mutants whose equivalence is rather a nuisance to judge; carelessness for these programs may lead to error proneness. Our experience, however, is that production programs present no special difficulties in this regard.

9. FURTHER APPLICATIONS OF MUTATION

9.1 Programming Tool. A tester specifies to an automatic mutation system:

- (1). a program,
- (2). test data,
- (3). a list of error operators to be applied.

The system generates and executes the required mutants on the test data, "killing" those which are judged incorrect vis a vis the execution of the subject program. The system also produces reports which the user may examine and use in subsequent attempts to eliminate mutants. This cycle may be viewed as a series of interactive sessions in which the user plays the role of an advocate who defends the program and the system plays the role of an adversary which asks questions of the form: why does your test data not distinguish this simple error?

If the mutation system also provides the user a pleasant runtime environment in which to write programs, the advocate-adversary relationship can be used to add an important dimension to the process of programming. Two of us have argued for the importance of "social" filters in the creative process [DLP]; mutation analysis applied during the

program design stage can be used to simulate an essential social process. We have observed in our own programming efforts and in the reported efforts of others, a tendency to communicate programs to others -- obviously the act of verbalizing ideas that had previously existed ethereally has a way of setting our intuitions (teachers have noted this phenomenon also). The typical exchange involves the programmer and a friendly but skeptical observer. As the programmer explains his code, the observer (even, if as is usually the case, he does not really understand the code) asks the sort of questions that one expects from a minimally attentive audience: "Why is that inequality not strict?", "Is that the same variable used at line 30?" In response to each such question, the programmer is forced to re-examine a fixed line of code and meet the objection -- he either justifies his decision to the observer, uncovers an error, or must admit that he really does not understand why the choice was made. In all three cases, the programmer receives valuable feedback that he is unlikely to have deduced by introspective analysis of the program.

The adversary role of a mutation system always forces the user into a careful and detailed review of his program and the design decisions made in constructing it. The mutations are like the minimally attentive observer who now and then chimes in with: "I don't believe that -- justify it!" Since it is a controlled form of "pointing" at the code which requires substantial cooperation from the user (his justification is a test case) such interactive use does in fact simulate an important aspect of the social process.

9.2 Project Management. Of the emerging approaches to software design, implementation and debugging -- however helpful they may be to programmers and local managers -- there are few that can be utilized throughout the project management hierarchy. Structured methods, program verification and restricted modularization are essentially qualitative, not quantitative, and managers should not be expected to understand the qualitative basis for the low-level decisions.

In addition to their primary function as evaluators of test data, mutation systems record a great deal of information which can be used to influence decision-making throughout the project hierarchy. Various management-oriented repackagings of the information relating to mutant failure percentages for each module (indicating how close the software is to being acceptable), who has responsibility for classifying which mutants as equivalent, and which mutants have yet to fail project management can:

- (1) reassign personnel to work on modules with low mutant failure rates,
- (2) pinpoint responsibility for modules which fail after acceptance,
- (3) use audits to force justification of why equivalent mutants exist,
- (4) monitor adherence to project PERT charts, and

- (5) offer rewards and incentives to programmers who achieve high mutant failure rates.

Obviously, the information reported to managers varies with the level of the manager, but a safe rule of thumb is that the higher in the organization a request for information originates, the less detailed is the expected response.

Project Manager's Report

The project manager periodically meets with the chief programmers to evaluate the project's status. Also, the assignment of personnel and evaluation of personnel performance are carried out at this level. A useful report for a manager at this level will contain:

- (1) the name of each module
- (2) the chief programmer responsible for each module
- (3) plots of mutant eliminations vs. time for each major submodule
- (4) summary statistics such as number and percentage of equivalent mutants,
- (5) the number and type of personnel assigned to each major submodule

Chief Programmer's Report

A chief programmer should be familiar with the actual coding of each submodule, although he is not always directly involved in the coding effort. He meets daily with his team. The type of information needed by the chief programmer would certainly encompass:

- (1)-(5) for the project manager
- (6) listings of equivalent mutants
- (7) logs assigning responsibility for classifying mutants as equivalent.

In addition to the goals outlined above, this information has the effect of suggesting possible additional mutant operators for a given submodule. Notice that the chief programmer assumes the responsibility for asking a programmer to justify mutant equivalence; assuming a postprocessor such as the one described in Section 8, these equivalent mutants should be largely non-trivial equivalences. A chief programmer may want to know for instance why it does not matter if a certain variable name can be changed without effect on the submodule, why the module is so insensitive to the mutation.

In the last analysis, it will be the chief programmer who determines that a given submodule has been acceptably tested and who will prepare evidence supporting his decision for the project manager.

Programmer's and Tester's Report

With the exception of the personnel reports, the programmer has access to all of the information supplied to the levels above him. He also has access to all listings, so can use the reporting mechanism to augment test data, augment mutant operators, classify equivalent mutants, and determine the adequacy of the test, all as described above.

9.3 Acceptance and Certification. The degree to which one has confidence in the competent programmer hypothesis and the coupling effect for the given set of mutant

operators determines the confidence that the mutant elimination percentage reflects the error-freeness of a program. However, in the absence of strong information in this regard, mutation analysis is an objective ranking device. Low elimination percentages are less desirable than high elimination percentages; furthermore, even though the boundary may be rather fuzzy, it is rather easy to reject obviously inadequate test data sets. This observation coupled with the fact that if all that is desired is an indication of the strength of a previously produced test data set then virtually no human interaction is required to produce the analysis leads one to consider the use of mutation analysis for software procurement testing.

Since acceptance testing should be the final stage of the development process, a buyer can specify at what point the testing begins. Assuming that the developer is using testing techniques with the sensitivity of mutation, the buyer can monitor progress. To evaluate the delivered software (or advertised software in the increasingly active mail market for small system software), one may specify contractually that the developer must present a convincing case that he is not delivering "rigged" tests -- one way of doing this is to specify a minimal mutant elimination percentage. Many options ensue. Software not passing this minimal certification may be rejected with significant financial penalty to the developer. In this case it is not essential that the developer use a mutation system to develop the tests. It is important to note that no more significance should be attached to the level of performance required for acceptance than for, say the third-party testing of refrigerators by a well-known certifying organization; the certification merely establishes the likelihood that the developer has spent considerable effort in testing his software. Thereafter, the buyer's confidence will more likely be affected by nontechnical issues, such as the developers performance on similar projects.

10. CONCLUDING REMARKS

A program passes a mutation test with a set of test data D if it behave correctly on D and each mutant either fails to work as specified or is equivalent to the program. When a program passes such a test, we are sure that it is free from simple errors. In order to insure that such a program is also free from complex errors, one must appeal to an empirical principle called the **coupling effect** which states that such a set of test data is so sensitive that non-equivalent (complex) mutants are also likely to fail on D . The conceptual justification for the coupling effect parallels the probabilistic arguments used to justify the single fault methods used to test logic circuits [Chang]. We have presented a combination of empirical evidence and plausibility arguments in support of the coupling effect. This leads to the metatheorem of mutation analysis:

If P passes mutation analysis then either

- (1) P is correct, or
- (2) P is radically incorrect.

The **Competent Programmer Hypothesis** states that experienced programmers tend to write programs that differ from correct ones by simple errors and hence possibility (2) of the metatheorem is rather unlikely.

In order that the mutation analysis technique be feasible, it is necessary that:

- (1) the set of simple mutants be small,
- (2) errors be reliably detected by the analysis, and
- (3) the question of equivalence be reducible to a small subproblem.

In the foregoing, we have presented our current knowledge with regard to these issues. Our experience has been encouraging. Even if the goals of mutation analysis are rather more optimistic than is warranted, the basis of a modelling strategy is emerging; it appears that it is possible to generate testable hypotheses concerning the programming process. We can only hope that future research by us and others will shed some light on this fascinating, important, but little understood, activity.

ACKNOWLEDGEMENTS

We are indebted to the members of the mutation research groups at Georgia Tech, Yale, and Berkeley. In particular, we would like to thank Jeannie Hanks, Douglas Baldwin, Jim Burns, Dan St. Andre, and Dan Hocking. We also thank Larry Yellowitz and an anonymous referee for their helpful comments.

REFERENCES

- [AU] A. Aho and J. Ullman, **The Theory of Parsing Translation and Compiling**, Vol 2: **Compiling**, Prentice-Hall, 1975.
- [BaS] D. Baldwin and F. Sayward, "Heuristics for Determining Equivalence of Program Mutations," Yale University, Department of Computer Science Research Report, No. 276, 1979.
- [BDLS] T. Budd, R.A.DeMillo, R.J. Lipton, and F.G. Sayward, "The Design of a Prototype Mutation System for Program Testing," **Proc. 1978 NCC, AFIPS Conference Record**, pp. 623-627.
- [BL1] T. Budd and R.J. Lipton, "Mutation Analysis of Decision Table Programs," **Proc. 1978 Conf. Information Sciences and Systems**, Johns Hopkins Univ., pp. 346-349.
- [BL2] T. Budd and R.J. Lipton, "Proving LISP Programs Using Test Data," **Digest for the Workshop on Software Testing and Test Documentation**, Fort Lauderdale, Fla.

1978, pp. 374-403.

- [Bur] J. Burns, "The stability of Test Data from Program Mutation," Digest for the Workshop on Software Testing and Test Documentation, Fort Lauderdale, Fla, 1978, pp. 324-334.
- [Cha] H.Y. Chang, **Fault Diagnosis of Digital Systems**, Wiley-Interscience, 1970.
- [DL] R.A. DeMillo and R.J. Lipton, "A Probabilistic Remark on Algebraic Program Testing," **Information Processing Letters**, Vol. 7(4), (June, 1978), pp.193-195.
- [DLP] R.A. DeMillo, R.J. Lipton and A.J. Perlis, "Social Processes and Proofs of Theorems and Programs," **CACM**, Vol 22(5), (May, 1979), pp. 271-280.
- [DLS1] R.A. DeMillo, R.J. Lipton and F.G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," **Computer**, April, 1978, pp. 34-41.
- [DLS2] R.A. DeMillo, R.J. Lipton and F.G. Sayward, "Program Mutation: A New Approach to Program Testing," **INFOTECH State of the Art Report on Software Testing**, Vol. 2, INFOTECH/SRA, 1979, pp. 107-127 [Note: also see commentaries in Volume 1].
- [Gel] M. Geller, "Test Data as an Aid in Proving Program Correctness," **CACM**, Vol 21(5), (May, 1978), pp. 368-375.
- [Gil] T. Gilb, **Software Metrics**, Winthrop, 1977.
- [Good] J. Goodenough, "A Survey of Program Testing Issues," in P. Wegner (editor), **Research Directions in Software Technology**, MIT Press, 1979, pp. 316-340.
- [GG] J. Goodenough and S. Gerhart, "Toward a Theory of Test Data Selection," **IEEE Trans. Software Engin.**, Vol SE-1, (June, 1975), pp. 156-173.
- [Ham] R.G. Hamlet, "Testing Programs with the Aid of a Compiler," **IEEE Trans. Software Engin.**, Vol. SE-3 (4), (July 1977), pp.
- [Hoa] C.A.R. Hoare, "Algorithm 65: FIND," **CACM**, Vol. 4(1), (January, 1961), p. 321.
- [How1] W.E. Howden, "Reliability of the Path Analysis Testing Strategy," **IEEE Trans. Software Engineering**, Vol. SE-2(3) (September, 1976), pp. 208-214.
- [How2] W.E. Howden, "An Evaluation of the Effectiveness of Symbolic Testing," **Software Practice and Experience**,

Volume 8, (1978), pp. 381-397.

- [Hua] J.C. Huang, "An Approach to Program Testing," **ACM Computing Surveys**, (September 1975), pp
- [KP] B.W. Kernhigan and P. Plauger, **The Elements of Programming Style**, McGraw-Hill, 1978 (Second Ed).
- [Kn] D.E.Knuth, "An Empirical Study of Fortran Programs," **Software Practice and Experience**, Vol. 1(2), (1971), pp. 105-134.
- [LMW] R.C. Linger, H.D. Mills and B.I. Witt, **Structured Programming Theory and Practice**, Addison-Weseley, 1979.
- [LS] R.J. Lipton and F.G. Sayward, "The Status of Research on Program Mutation," Digest of the Workshop on Software Testing and Test Documentation, Fort Lauderdale, Fla, 1978, pp. 355-373.
- [Man] Z. Manna, **The Mathematical Theory of Computation**, McGraw-Hill, 1974.
- [Nau] P. Nauer, "Programming by Action Clusters," **BIT**, Vol. 9, (1969), pp. 250-258.
- [OF1] L.J. Osterweil and L.D. Fosdick, "Experience with DAVE-- A Fortran Program Analyzer, **Proc. 1976 NCC, AFIPS Conference Record**, pp 909-915.
- [OF2] L.J. Osterweil and L.D. Fosdick, "Data Flow Analysis as an Aid in Documentation, Assertion Generation, Validation and Error Detection, University of Colorado, Department of Computer Science, Technical Report No. CU-CS-055-74, 1974.
- [OW] T.J. Ostrand and E.J. Weyuker, "Remarks on the Theory of Test Data Selection," Digest for Workshop on Software Testing and Test Documentation, Fort Lauderdale Fla, 1978, pp. 1-18.
- [Sch] M. Schaefer, **A Mathematical Theory of Global Program Optimization**, Prentice-Hall, 1973.
- [TRW] T.A. Thayer, M. Lipow, E.C. Nelson, **Software Reliability**, North-Holland, 1978.
- [WCC] L.J White, E. Cohen, and B. Chandrasekaran, "A Domain Strategy for Computer Program Testing," Ohio State University, Department of Computer and Information Science, Report No. OSU-CISRC-TR-78-4, 1978.
- [Wir] N. Wirth, "PL360, A programming Language for the 360 Computer," **JACM**, Vol. 15(1), (January 1968), pp. 37-74.

[You] E.A. Youngs, "Human Errors in Programming," **International Journal of Man-Machine Studies**, Volume 6 (1974), pp. 361-376.

APPENDIX

CMS.1 SESSION SCRIPT

WELCOME TO THE COBOL PILOT MUTATION SYSTEM
PLEASE ENTER THE NAME OF THE COBOL PROGRAM FILE:>LOG-CHANGES
DO YOU WANT TO PURGE WORKING FILES FOR A FRESH RUN ?>YES
PARSING PROGRAM
SAVING INTERNAL FORM
WHAT PERCENTAGE OF MUTANTS DO YOU WANT TO CREATE?>100
CREATING MUTANT DESCRIPTOR RECORDS
PRE-RUN PHASE
DO YOU WANT TO SUBMIT A TEST CASE ? >PROGRAM
1 IDENTIFICATION DIVISION.
2 PROGRAM-ID. POQAACA.
3 AUTHOR. CPT R W MOREHEAD.
4 INSTALLATION. HQS USACSC.
5 DATE-WRITTEN. OCT 1973.
6 REMARKS.
7 THIS PROGRAM PRINTS OUT A LIST OF CHANGES IN THE ETF.
8 ALL ETF CHANGES WERE PROCESSED PRIOR TO THIS PROGRAM. THE
9 OLD ETF AND THE NEW ETF ARE THE INPUTS. BUT THERE IS NO
10 FURTHER PROCESSING OF THE ETF HERE. THE ONLY OUTPUT IS A
11 LISTING OF THE ADDS, CHANGES, AND DELETES. THIS PROGRAM IS
12 FOR HQ USE ONLY AND HAS NO APPLICATION IN THE FIELD.
13 *****
14 MODIFIED FOR TESTING UNDER CPMS BY ALLEN ACREE
15 JULY, 1979.
16 ENVIRONMENT DIVISION.
17 CONFIGURATION SECTION.
18 SOURCE-COMPUTER. PRIME.
19 OBJECT-COMPUTER. PRIME.
20 INPUT-OUTPUT SECTION.
21 FILE-CONTROL.
22 SELECT OLD-ETF ASSIGN INPUT4.
23 SELECT NEW-ETF ASSIGN INPUT8.
24 SELECT PRNTR ASSIGN TO OUTPUT9.
25 DATA DIVISION.
26 FILE SECTION.
27 FD OLD-ETF
28 RECORD CONTAINS 80 CHARACTERS
29 LABEL RECORDS ARE STANDARD
30 DATA RECORD IS OLD-REC.
31 01 OLD-REC.
32 03 FILLER PIC X.
33 03 OLD-KEY PIC X(12).
34 03 FILLER PIC X(67).
35 FD NEW-ETF
36 RECORD CONTAINS 80 CHARACTERS
37 LABEL RECORDS ARE STANDARD
38 DATA RECORD IS NEW-REC.
39 01 NEW-REC.
40 03 FILLER PIC X.
41 03 NEW-KEY PIC X(12).

```

42      03 FILLER                                PIC X(67).
43  FD  PRNTR
44      RECORD CONTAINS 40 CHARACTERS
45      LABEL RECORDS ARE OMITTED
46      DATA RECORD IS PRNT-LINE.
47  01  PRNT-LINE                                PIC X(40).
48      WORKING-STORAGE SECTION.
49  01  PRNT-WORK-AREA.
50      03 LINE1                                PIC X(30).
51      03 LINE2                                PIC X(30).
52      03 LINE3                                PIC X(20).
53  01  PRNT-OUT-OLD.
54      03 WS-LN-1.
55          05 FILLER                            PIC X VALUE SPACE.
56          05 FILLER                            PIC XXXX VALUE 'O '.
57          05 LN1                                PIC X(30).
58          05 FILLER                            PIC XXX VALUE SPACES.
59      03 WS-LN-2.
60          05 FILLER                            PIC X VALUE SPACE.
61          05 FILLER                            PIC XXXX VALUE 'L '.
62          05 LN2                                PIC X(30).
63          05 FILLER                            PIC XXX VALUE SPACES.
64      03 WS-LN-3.
65          05 FILLER                            PIC X VALUE SPACE.
66          05 FILLER                            PIC XXXX VALUE 'D '.
67          05 LN3                                PIC X(20).
68          05 FILLER                            PIC XXX VALUE SPACE.
69  01  PRNT-NEW-OUT.
70      03 NEW-LN-1.
71          05 FILLER                            PIC XXXXX VALUE ' N '.
72          05 N-LN1                              PIC X(30).
73          05 FILLER                            PIC XXX VALUE SPACE.
74      03 NEW-LN-2.
75          05 FILLER                            PIC XXXXX VALUE ' E '.
76          05 N-LN2                              PIC X(30).
77          05 FILLER                            PIC XXX VALUE SPACES.
78      03 NEW-LN-3.
79          05 FILLER                            PIC XXXXX VALUE ' W '.
80          05 N-LN3                              PIC X(20).
81          05 FILLER                            PIC XXX VALUE SPACES.
82  PROCEDURE DIVISION.
83  0100-OPENS.
84      OPEN INPUT OLD-ETF NEW-ETF.
85      OPEN OUTPUT PRNTR.
86  0110-OLD-READ.
87      READ OLD-ETF AT END GO TO 0160-OLD-EOF.
88  0120-NEW-READ.
89      READ NEW-ETF AT END GO TO 0170-NEW-EOF.
90  0130-COMPARES.
91      IF OLD-KEY = NEW-KEY
92          NEXT SENTENCE
93      ELSE GO TO 0140-CK-ADD-DEL.
94      IF OLD-REC = NEW-REC
95          GO TO 0110-OLD-READ.
96      MOVE OLD-REC TO PRNT-WORK-AREA.
97      PERFORM 0210-OLD-WRT THRU 0210-EXIT.

```

```

98      MOVE NEW-REC TO PRNT-WORK-AREA.
99      PERFORM 0200-NW-WRT THRU 0200-EXIT.
100     GO TO 0110-OLD-READ.
101 0140-CK-ADD-DEL.
102     IF OLD-KEY > NEW-KEY
103         MOVE NEW-REC TO PRNT-WORK-AREA
104         PERFORM 0200-NW-WRT THRU 0200-EXIT
105         GO TO 0120-NEW-READ
106     ELSE GO TO 0150-CK-ADD-DEL.
107 0150-CK-ADD-DEL.
108     MOVE OLD-REC TO PRNT-WORK-AREA.
109     PERFORM 0210-OLD-WRT THRU 0210-EXIT.
110     READ OLD-ETF AT END
111         MOVE NEW-REC TO PRNT-WORK-AREA
112         PERFORM 0200-NW-WRT THRU 0200-EXIT
113         GO TO 0160-OLD-EOF.
114     GO TO 0130-COMPARES.
115 0160-OLD-EOF.
116     READ NEW-ETF AT END GO TO 0180-EOJ.
117     MOVE NEW-REC TO PRNT-WORK-AREA.
118     PERFORM 0200-NW-WRT THRU 0200-EXIT.
119     GO TO 0160-OLD-EOF.
120 0170-NEW-EOF.
121     MOVE OLD-REC TO PRNT-WORK-AREA.
122     PERFORM 0210-OLD-WRT THRU 0210-EXIT.
123     READ OLD-ETF AT END GO TO 0180-EOJ.
124     GO TO 0170-NEW-EOF.
125 0180-EOJ.
126     CLOSE OLD-ETF NEW-ETF PRNTR.
127     STOP RUN.
128 0200-NW-WRT.
129     MOVE LINE1 TO N-LN1.
130     MOVE LINE2 TO N-LN2.
131     MOVE LINE3 TO N-LN3.
132     WRITE PRNT-LINE FROM NEW-LN-1 AFTER ADVANCING 2.
133     WRITE PRNT-LINE FROM NEW-LN-2 AFTER ADVANCING 1.
134     WRITE PRNT-LINE FROM NEW-LN-3 AFTER ADVANCING 1.
135 0200-EXIT.
136     EXIT.
137 0210-OLD-WRT.
138     MOVE LINE1 TO LN1.
139     MOVE LINE2 TO LN2.
140     MOVE LINE3 TO LN3.
141     WRITE PRNT-LINE FROM WS-LN-1 AFTER ADVANCING 2.
142     WRITE PRNT-LINE FROM WS-LN-2 AFTER ADVANCING 1.
143     WRITE PRNT-LINE FROM WS-LN-3 AFTER ADVANCING 1.
144 0210-EXIT.
145     EXIT.
>YES

```

A test case for this program is a pair of input files. In CMS.1 these may be created outside the system and referenced by name, or may be entered "on the fly".

WHERE IS OLD-ETF?

WHERE IS NEW-ETF?

OLD-ETF AS USED BY THE PROGRAM

I123456789012IIIIIIIIIOJJJJJJJKKKKKKKKKKLLLLLLLLLNNNNNNNNNNNBBBBBBBBBBGGGGG
J23456789012YYYYYYYYYGGGGGGGGGGFFFFFFFFFFODDDDDDDDDSSSSSSSSSSXXXXXXXXXXEEEE

NEW-ETF AS USED BY THE PROGRAM

[illegible]

PRNTR AS USED BY THE PROGRAM

O I123456789012IIIIIIIIIIIOJJJJJJ
L JJJKKKKKKKKKKLLLLLLLLLLLLNNNNNNNN
D NNNBBBBBBBBBBBBGGGGGGGG

```
N I133456789012000000000000000000  
E 00000000000000000000000000000000  
W 00000000000000000000
```

```
O J234567890123YYYYYYYYYGGGGGGG
L GGGFFFFFFFFFFFF0DDDDDDDDDDSSSSSSS
D SSSXXXXXXXXXXXXEEEEEEE
```

```
N J234567890123YYYYYYYYYGGGGGGG
E GGGFFFFFFFFFDDDDDDDDSSSSSS
W SSSXXXXXXXXXXEEEEEEE
```

```
N 345678901234UUUUUUUUUUHHHHHHH
E HHHGGGGGGGGGGDDDDDDDDDDSSSSSSS
W SSSEEEEEEEEEFAAAAAAA
```

```

THE PROGRAM TOOK 84 STEPS
IS THIS TEST CASE ACCEPTABLE ? >YES
DO YOU WANT TO SUBMIT A TEST CASE ? >NO
MUTATION PHASE
WHAT NEW MUTANT TYPES ARE TO BE CONSIDERED ? >SELECT

```

ENTER THE NUMBERS OF THE MUTANT TYPES YOU WANT TO TURN ON AT THIS TIME.

```

4      **** INSERT FILLER TYPE      ****
5      **** FILLER SIZE ALTERATION TYPE ****
6      **** ELEMENTARY ITEM REVERSAL TYPE ****
7      **** FILE REFERENCE ALTERATION TYPE ****
8      **** STATEMENT DELETION TYPE ****
10     **** PERFORM --> GO TO TYPE ****
11     **** THEN - ELSE REVERSAL TYPE ****
12     **** STOP STATEMENT SUBSTITUTION TYPE ****
13     **** THRU CLAUSE EXTENSION TYPE ****
14     **** TRAP STATEMENT REPLACEMENT TYPE ****
20     **** LOGICAL OPERATOR REPLACEMENT TYPE ****
21     **** SCALAR FOR SCALAR REPLACEMENT ****

```


TOTALS

1195 37 96.90

DO YOU WANT TO SEE THE LIVE MUTANTS?>>YES

FOR EACH MUTANT :

HIT RETURN TO CONTINUE. TYPE 'STOP' TO STOP.

TYPE 'EQUIV' TO JUDGE THE MUTANT EQUIVALENT.

**** INSERT FILLER TYPE ****

MUTANT NUMBER 12

A FILLER OF LENGTH ONE HAS BEEN INSERTED AFTER
THE ITEM WHICH STARTS ON LINE 52
ITS LEVEL NUMBER IS 3

>

MUTANT NUMBER 13

A FILLER OF LENGTH ONE HAS BEEN INSERTED AFTER
THE ITEM WHICH STARTS ON LINE 53
ITS LEVEL NUMBER IS 3

>

MUTANT NUMBER 29

A FILLER OF LENGTH ONE HAS BEEN INSERTED AFTER
THE ITEM WHICH STARTS ON LINE 69
ITS LEVEL NUMBER IS 3

>

**** FILLER SIZE ALTERATION TYPE ****

MUTANT NUMBER 54

THE FILLER ON LINE 58 HAS HAD ITS SIZE DECREMENTED BY ONE.

>

MUTANT NUMBER 55

THE FILLER ON LINE 58 HAS HAD ITS SIZE INCREMENTED BY ONE.

>

MUTANT NUMBER 60

THE FILLER ON LINE 63 HAS HAD ITS SIZE DECREMENTED BY ONE.

>

MUTANT NUMBER 61

THE FILLER ON LINE 63 HAS HAD ITS SIZE INCREMENTED BY ONE.

>

MUTANT NUMBER 66

THE FILLER ON LINE 68 HAS HAD ITS SIZE DECREMENTED BY ONE.

>

MUTANT NUMBER 67

THE FILLER ON LINE 68 HAS HAD ITS SIZE INCREMENTED BY ONE.

>

MUTANT NUMBER 70

THE FILLER ON LINE 73 HAS HAD ITS SIZE DECREMENTED BY ONE.

>

MUTANT NUMBER 71

THE FILLER ON LINE 73 HAS HAD ITS SIZE INCREMENTED BY ONE.

>

MUTANT NUMBER 74

THE FILLER ON LINE 77 HAS HAD ITS SIZE DECREMENTED BY ONE.

>

MUTANT NUMBER 75

THE FILLER ON LINE 77 HAS HAD ITS SIZE INCREMENTED BY ONE.

>

MUTANT NUMBER 78

THE FILLER ON LINE 81 HAS HAD ITS SIZE DECREMENTED BY ONE.

>

MUTANT NUMBER 79

THE FILLER ON LINE 81 HAS HAD ITS SIZE INCREMENTED BY ONE.

>

**** STATEMENT DELETION TYPE ****

MUTANT NUMBER 126

ON LINE 106 THE STATEMENT:

GO TO 0150-CK-ADD-DEL

HAS BEEN DELETED.

>

**** LOGICAL OPERATOR REPLACEMENT TYPE ****

MUTANT NUMBER 296

ON LINE 102 THE STATEMENT:

IF OLD-KEY > NEW-KEY

HAS BEEN CHANGED TO:

IF OLD-KEY NOT < NEW-KEY

>

**** SCALAR FOR SCALAR REPLACEMENT ****

MUTANT NUMBER 300

ON LINE 87 THE STATEMENT:

READ OLD-ETF AT END ...

HAS BEEN CHANGED TO:

READ OLD-ETF INTO NEW-REC AT END ...

>

MUTANT NUMBER 301

ON LINE 87 THE STATEMENT:

READ OLD-ETF AT END ...

HAS BEEN CHANGED TO:

READ OLD-ETF INTO PRNT-WORK-AREA AT END ...

>

MUTANT NUMBER 311

ON LINE 89 THE STATEMENT:

READ NEW-ETF AT END ...

HAS BEEN CHANGED TO:

READ NEW-ETF INTO PRNT-WORK-AREA AT END ...

>

MUTANT NUMBER 629

ON LINE 110 THE STATEMENT:

READ OLD-ETF AT END ...

HAS BEEN CHANGED TO:

READ OLD-ETF INTO PRNT-WORK-AREA AT END ...

>

MUTANT NUMBER 682

ON LINE 116 THE STATEMENT:

READ NEW-ETF AT END ...

HAS BEEN CHANGED TO:

READ NEW-ETF INTO OLD-REC AT END ...

>

MUTANT NUMBER 683

ON LINE 116 THE STATEMENT:

READ NEW-ETF AT END ...

HAS BEEN CHANGED TO:

READ NEW-ETF INTO PRNT-WORK-AREA AT END ...

>

MUTANT NUMBER 684

ON LINE 116 THE STATEMENT:

READ NEW-ETF AT END ...

HAS BEEN CHANGED TO:

READ NEW-ETF INTO PRNT-OUT-OLD AT END ...

>

MUTANT NUMBER 685

ON LINE 116 THE STATEMENT:

READ NEW-ETF AT END ...

HAS BEEN CHANGED TO:

READ NEW-ETF INTO WS-LN-1 AT END ...

>

MUTANT NUMBER 686

ON LINE 116 THE STATEMENT:

READ NEW-ETF AT END ...

HAS BEEN CHANGED TO:

READ NEW-ETF INTO WS-LN-2 AT END ...

>

MUTANT NUMBER 687

ON LINE 116 THE STATEMENT:

READ NEW-ETF AT END ...

HAS BEEN CHANGED TO:

READ NEW-ETF INTO WS-LN-3 AT END ...

>

MUTANT NUMBER 780

ON LINE 123 THE STATEMENT:

READ OLD-ETF AT END ...

HAS BEEN CHANGED TO:

READ OLD-ETF INTO NEW-REC AT END ...

>

MUTANT NUMBER 781

ON LINE 123 THE STATEMENT:

READ OLD-ETF AT END ...

HAS BEEN CHANGED TO:

READ OLD-ETF INTO PRNT-WORK-AREA AT END ...

>

MUTANT NUMBER 786

ON LINE 123 THE STATEMENT:

READ OLD-ETF AT END ...

HAS BEEN CHANGED TO:

READ OLD-ETF INTO PRNT-NEW-OUT AT END ...

>

MUTANT NUMBER 787

ON LINE 123 THE STATEMENT:

READ OLD-ETF AT END ...

HAS BEEN CHANGED TO:

READ OLD-ETF INTO NEW-LN-1 AT END ...

>

MUTANT NUMBER 788

ON LINE 123 THE STATEMENT:

READ OLD-ETF AT END ...

HAS BEEN CHANGED TO:

READ OLD-ETF INTO NEW-LN-2 AT END ...

>

MUTANT NUMBER 789

ON LINE 123 THE STATEMENT:

READ OLD-ETF AT END ...

HAS BEEN CHANGED TO:

READ OLD-ETF INTO NEW-LN-3 AT END ...

>

MUTANT NUMBER 814

ON LINE 129 THE STATEMENT:

MOVE LINE1 TO N-LN1

HAS BEEN CHANGED TO:

MOVE NEW-REC TO N-LN1

>

MUTANT NUMBER 817

ON LINE 129 THE STATEMENT:

MOVE LINE1 TO N-LN1

HAS BEEN CHANGED TO:

MOVE PRNT-WORK-AREA TO N-LN1

>

MUTANT NUMBER 974

ON LINE 138 THE STATEMENT:

MOVE LINE1 TO LN1

HAS BEEN CHANGED TO:

MOVE OLD-REC TO LN1

>

MUTANT NUMBER 979

ON LINE 138 THE STATEMENT:

MOVE LINE1 TO LN1

HAS BEEN CHANGED TO:

MOVE PRNT-WORK-AREA TO LN1

>

LOOP OR HALT ? >HALT

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER 14 GIT-ICS-79/08	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) 6 Mutation Analysis 12, 92	5. TYPE OF REPORT & PERIOD COVERED 9 Final Report	
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) A. T. Acree, T. J. Budd, R. A. DeMillo, R. J. Lipton, F. G. Sayward	8. CONTRACT OR GRANT NUMBER(s) DAAG29-78-0121	
9. PERFORMING ORGANIZATION NAME AND ADDRESS School of Information and Computer Science Georgia Institute of Technology Atlanta, Georgia 30332 410 044		10. PROGRAM ELEMENT PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS 11		12. REPORT DATE September 1979
		13. NUMBER OF PAGES 86
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) US Army Research Office PO Box 12211 Research Triangle Park, N.C. 27709 18 ARD		15. SECURITY CLASS. (of this report) 15a. DECLASSIFICATION DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) Approved for Published Releases; Distribution Unlimited 19 15950,4-A-EL		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report) 15 N00014-79-C-0231, DAAG29-78-G-0121		
18. SUPPLEMENTARY NOTES The findings of this report are not to be construed as an official Department of the Army position, unless so designated by other authorized document.		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) 10 Allen T. /Acree, Timothy A. /Budd, Richard A. /DeMillo, Richard J. /Lipton Frederick G. /Sayward		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) A new type of software test, called mutation analysis, is introduced. A method of applying mutation analysis is described, and the design of several existing automated systems for applying mutation analysis to Fortran and Cobol programs is sketched. These systems have been the means for preliminary studies of the efficiency of mutation analysis and of the relationship between mutation and other systematic testing techniques. The results of several experiments to determine the effectiveness of mutation analysis are described, and examples are (continued - over)		

20. Abstract (continued)

presented to illustrate the way in which the technique can be used to detect a wide class of errors, including many previously defined and studied in the literature. Finally, a number of empirical studies are suggested, the results of which may add confidence to the outcome of the mutation analysis of a program.